Spark 编程指南

简体中文版

Andy Ai

Published
with GitBook



目錄

Spark 编程指南简体中文版

Copyright

License

快速上手

本节课程提供一个使用 Spark 的快速介绍，首先我们使用 Spark 的交互式 shell(用 Python 或 Scala) 介绍它的 API。当演示如何在 Java, Scala 和 Python 写独立的程序时，看编程指南里完整的参考。

依照这个指南，首先从 Spark 网站下载一个 Spark 发行包。因为我们不会使用 HDFS，你可以下载任何 Hadoop 版本的包。

- Spark Shell
- 独立应用程序
- 开始翻滚吧!

使用 Spark Shell

基础

Spark 的 shell 作为一个强大的交互式数据分析工具，提供了一个简单的方式来学习 API。它可以使用 Scala(在 Java 虚拟机上运行现有的 Java 库的一个很好方式) 或 Python。在 Spark 目录里使用下面的方式开始运行：

```
./bin/spark-shell
```

Spark 最主要的抽象是叫Resilient Distributed Dataset(RDD) 的弹性分布式集合。RDDs 可以使用 Hadoop InputFormats(例如 HDFS 文件)创建，也可以从其他的 RDDs 转换。让我们在 Spark 源代码目录从 README 文本文件中创建一个新的 RDD。

```
scala> val textFile = sc.textFile("README.md")
textFile: spark.RDD[String] = spark.MappedRDD@2ee9b6e3
```

RDD 的 actions 从 RDD 中返回值，transformations 可以转换成一个新 RDD 并返回它的引用。让我们开始使用几个操作：

```
scala> textFile.count() // RDD 的数据条数
res0: Long = 126

scala> textFile.first() // RDD 的第一行数据
res1: String = # Apache Spark
```

现在让我们使用一个 transformation，我们将使用 filter 在这个文件里返回一个包含子数据集的新 RDD。

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: spark.RDD[String] = spark.FilteredRDD@7dd4af09
```

我们可以把 actions 和 transformations 链接在一起：

```
scala> textFile.filter(line => line.contains("Spark")).count() // 有多少行包括 "Spark"?
res3: Long = 15
```

更多 RDD 操作

RDD actions 和 transformations 能被用在更多的复杂计算中。比方说，我们想要找到一行中最多的单词数量：

```
scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
res4: Long = 15
```

首先将行映射成一个整型数值产生一个新 RDD。 在这个新的 RDD 上调用 `reduce` 找到行中最大的个数。 `map` 和 `reduce` 的参数是 Scala 的函数串(闭包)，并且可以使用任何语言特性或者 Scala/Java 类库。例如，我们可以很方便地调用其他的函数声明。 我们使用 `Math.max()` 函数让代码更容易理解：

```
scala> import java.lang.Math
import java.lang.Math

scala> textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))
res5: Int = 15
```

Hadoop 流行的一个通用的数据流模式是 MapReduce。Spark 能很容易地实现 MapReduce：

```
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).
wordCounts: spark.RDD[(String, Int)] = spark.ShuffledAggregatedRDD@71f027b8
```

这里，我们结合 flatMap, map 和 reduceByKey 来计算文件里每个单词出现的数量，它的结果是包含一组(String, Int) 键值对的 RDD。我们可以使用 [collect] 操作在我们的 shell 中收集单词的数量：

```
scala> wordCounts.collect()
res6: Array[(String, Int)] = Array((means,1), (under,2), (this,3), (Because,1), (Python,2
```

缓存

Spark 支持把数据集拉到集群内的内存缓存中。当要重复访问时这是非常有用的，例如当我们在一个小的热(hot)数据集中查询，或者运行一个像网页搜索排序这样的重复算法。作为一个简单的例子，让我们把 `linesWithSpark` 数据集标记在缓存中：

```
scala> linesWithSpark.cache()
res7: spark.RDD[String] = spark.FilteredRDD@17e51082

scala> linesWithSpark.count()
res8: Long = 15

scala> linesWithSpark.count()
res9: Long = 15
```

缓存 100 行的文本文件来研究 Spark 这看起来很傻。真正让人感兴趣的部分是我们可以在非常大型的数据集中使用同样的函数，甚至在 10 个或者 100 个节点中交叉计算。你同样可以使用 `bin/spark-shell` 连接到一个 cluster 来替换掉编程指南中的方法进行交互操作。

独立应用程序

现在假设我们想要使用 Spark API 写一个独立的应用程序。我们将通过使用 Scala(用 SBT), Java(用 Maven) 和 Python 写一个简单的应用程序来学习。

我们用 Scala 创建一个非常简单的 Spark 应用程序。如此简单，事实上它的名字叫 `SimpleApp.scala` :

```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // 应该是你系统上的某些文件
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

这个程序仅仅是在 Spark README 中计算行里面包含 'a' 和包含 'b' 的次数。你需要注意将 `YOUR_SPARK_HOME` 替换成你已经安装 Spark 的路径。不像之前的 Spark Shell 例子，这里初始化了自己的 SparkContext，我们把 SparkContext 初始化作为程序的一部分。

我们通过 SparkContext 的构造函数参入 SparkConf 对象，这个对象包含了一些关于我们程序的信息。

我们的程序依赖于 Spark API，所以我们需要包含一个 sbt 文件文件， `simple.sbt` 解释了 Spark 是一个依赖。这个文件还要补充 Spark 依赖于一个 repository :

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.0"
```

要让 sbt 正确工作，我们需要把 `SimpleApp.scala` 和 `simple.sbt` 按照标准的文件目录结构布局。上面的做好之后，我们可以把程序的代码创建成一个 JAR 包。然后使用 `spark-submit` 来运行我们的程序。

```
# Your directory layout should look like this
$ find .
.
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala

# Package a jar containing your application
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.10/simple-project_2.10-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local[4] \
  target/scala-2.10/simple-project_2.10-1.0.jar
...
Lines with a: 46, Lines with b: 23
```

开始翻滚吧!

祝贺你成功运行你的第一个 Spark 应用程序!

- 要深入了解 API，可以从Spark编程指南开始，或者从其他的组件开始，例如：Spark Streaming。
- 要让程序运行在集群(cluster)上，前往部署概论。
- 最后，Spark 在 `examples` 文件目录里包含了 Scala, Java 和 Python 的几个简单的例子，你可以直接运行它们：

```
# For Scala and Java, use run-example:
./bin/run-example SparkPi

# For Python examples, use spark-submit directly:
./bin/spark-submit examples/src/main/python/pi.py
```

概论

宏观上说，每个 Spark 应用程序都由一个驱动程序*(driver programe)*构成，驱动程序在集群上运行用户的 `main` 函数来执行各种各样的并行操作*(parallel operations)*。Spark 的主要抽象是提供一个弹性分布式数据集*(RDD resilient distributed dataset)*，RDD 是指能横跨集群所有节点进行并行计算的分区元素集合。RDD 可以从 Hadoop 文件系统中的一个文件中创建而来(或其他 Hadoop 支持的文件系统)，或者从一个已有的 Scala 集合转换得到。用户可以要求 Spark 将 RDD 持久化*(persist)*到内存中，来让它在并行计算中高效地重用。最后，RDD 能从节点失败中自动地恢复过来。

Spark 的第二个抽象是共享变量*(shared variables)*，共享变量能被运行在并行计算中。默认情况下，当 Spark 运行一个并行函数时，这个并行函数会作为一个任务集在不同的节点上运行，它会把函数里使用的每个变量都复制搬运到每个任务中。有时，一个变量需要被共享到交叉任务中或驱动程序和任务之间。Spark 支持 2 种类型的共享变量：广播变量*(broadcast variables)*，用来在所有节点的内存中缓存一个值；累加器(accumulators)，仅仅只能执行“添加(added)”操作，例如：记数器(counters)和求和(sums)。

这个指南会在 Spark 支持的所有语言中演示它的每一个特征。可以非常简单地从一个 Spark 交互式 shell 开始 -—— `bin/spark-shell` 开始一个 Scala shell，或 `bin/pyspark` 开始一个 Python shell。

- 引入 Spark
- 初始化 Spark
- Spark RDDs
- 共享变量
- 从这里开始

引入 Spark

Spark 1.2.0 使用 Scala 2.10 写应用程序，你需要使用一个兼容的 Scala 版本(例如：2.10.X)。

写 Spark 应用程序时，你需要添加 Spark 的 Maven 依赖，Spark 可以通过 Maven 中心仓库来获得：

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.2.0
```

另外，如果你希望访问 HDFS 集群，你需要根据你的 HDFS 版本添加 `hadoop-client` 的依赖。一些公共的 HDFS 版本 tags 在第三方发行页面中被列出。

```
groupId = org.apache.hadoop
artifactId = hadoop-client
version = <your-hdfs-version>
```

最后，你需要导入一些 Spark 的类和隐式转换到你的程序，添加下面的行就可以了：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

初始化 Spark

Spark 编程的第一步是需要创建一个 SparkContext 对象，用来告诉 Spark 如何访问集群。在创建 `SparkContext` 之前，你需要构建一个 SparkConf 对象， SparkConf 对象包含了一些你应用程序的信息。

```
val conf = new SparkConf().setAppName(appName).setMaster(master)
new SparkContext(conf)
```

`appName` 参数是你程序的名字，它会显示在 cluster UI 上。 `master` 是 Spark, Mesos 或 YARN 集群的 URL，或运行在本地模式时，使用专用字符串 “local”。在实践中，当应用程序运行在一个集群上时，你并不想要把 `master` 硬编码到你的程序中，你可以用 spark-submit 启动你的应用程序的时候传递它。然而，你可以在本地测试和单元测试中使用 “local” 运行 Spark 进程。

使用 Shell

在 Spark shell 中，有一个专有的 SparkContext 已经为你创建好。在变量中叫做 `sc` 。你自己创建的 SparkContext 将无法工作。可以用 `--master` 参数来设置 SparkContext 要连接的集群，用 `--jars` 来设置需要添加到 classpath 中的 JAR 包，如果有多个 JAR 包使用逗号分割符连接它们。例如：在一个拥有 4 核的环境上运行 `bin/spark-shell` ，使用：

```
$ ./bin/spark-shell --master local[4]
```

或在 classpath 中添加 `code.jar` ，使用：

```
$ ./bin/spark-shell --master local[4] --jars code.jar
```

执行 `spark-shell --help` 获取完整的选项列表。在这之后，调用 `spark-shell` 会比 spark-submit 脚本更为普遍。

弹性分布式数据集 (RDDs)

Spark 核心的概念是 *Resilient Distributed Dataset (RDD)*：一个可并行操作的有容错机制的数据集合。有 2 种方式创建 RDDs：第一种是在你的驱动程序中并行化一个已经存在的集合；另外一种是引用一个外部存储系统的数据集，例如共享的文件系统，HDFS，HBase或其他 Hadoop 数据格式的数据源。

- 并行集合
- 外部数据集
- RDD 操作 <<<<<<< HEAD
- 传递函数到 Spark
- 使用键值对
- Transformations
- Actions
- # RDD持久化
    - 传递函数到 Spark
    - 使用键值对
    - Transformations
    - Actions
- RDD 持久化

  > > > > > > > master

并行集合

并行集合 (*Parallelized collections*) 的创建是通过在一个已有的集合(Scala `Seq`)上调用 SparkContext 的 `parallelize` 方法实现的。集合中的元素被复制到一个可并行操作的分布式数据集中。例如，这里演示了如何在一个包含 1 到 5 的数组中创建并行集合：

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

一旦创建完成，这个分布式数据集(`distData`)就可以被并行操作。例如，我们可以调用 `distData.reduce((a, b) => a + b)` 将这个数组中的元素相加。我们以后再描述在分布式上的一些操作。

并行集合一个很重要的参数是切片数(*slices*)，表示一个数据集切分的份数。Spark 会在集群上为每一个切片运行一个任务。你可以在集群上为每个 CPU 设置 2-4 个切片(slices)。正常情况下，Spark 会试着基于你的集群状况自动地设置切片的数目。然而，你也可以通过 `parallelize` 的第二个参数手动地设置(例如： `sc.parallelize(data, 10)`)。

外部数据集

Spark 可以从任何一个 Hadoop 支持的存储源创建分布式数据集，包括你的本地文件系统，HDFS，Cassandra，HBase，Amazon S3等。 Spark 支持文本文件(text files)，SequenceFiles 和其他 Hadoop InputFormat。

文本文件 RDDs 可以使用 SparkContext 的 `textFile` 方法创建。 在这个方法里传入文件的 URI (机器上的本地路径或 `hdfs://` ， `s3n://` 等)，然后它会将文件读取成一个行集合。这里是一个调用例子：

```
scala> val distFile = sc.textFile("data.txt")
distFile: RDD[String] = MappedRDD@1d4cee08
```

一旦创建完成， `distFiile` 就能做数据集操作。例如，我们可以用下面的方式使用 `map` 和 `reduce` 操作将所有行的长度相加： `distFile.map(s => s.length).reduce((a, b) => a + b)` 。

注意，Spark 读文件时：

- 如果使用本地文件系统路径，文件必须能在 work 节点上用相同的路径访问到。要么复制文件到所有的 workers，要么使用网络的方式共享文件系统。
- 所有 Spark 的基于文件的方法，包括 `textFile` ，能很好地支持文件目录，压缩过的文件和通配符。例如，你可以使用 `textFile("/my/文件目录")` ， `textFile("/my/文件目录/*.txt")` 和 `textFile("/my/文件目录/*.gz")` 。
- `textFile` 方法也可以选择第二个可选参数来控制切片(*slices*)的数目。默认情况下，Spark 为每一个文件块(HDFS 默认文件块大小是 64M)创建一个切片(*slice*)。但是你也可以通过一个更大的值来设置一个更高的切片数目。注意，你不能设置一个小于文件块数目的切片值。

除了文本文件，Spark 的 Scala API 支持其他几种数据格式：

- `SparkContext.wholeTextFiles` 让你读取一个包含多个小文本文件的文件目录并且返回每一个(filename, content)对。与 `textFile` 的差异是：它记录的是每个文件中的每一行。
- 对于 SequenceFiles，可以使用 SparkContext 的 `sequenceFile[K, V]` 方法创建，K 和 V 分别对应的是 key 和 values 的类型。像 IntWritable 与 Text 一样，它们必须是 Hadoop 的 Writable 接口的子类。另外，对于几种通用的 Writables，Spark 允许你指定原生类型来替代。例如： `sequenceFile[Int, String]` 将会自动读取 IntWritables 和 Text。
- 对于其他的 Hadoop InputFormats，你可以使用 `SparkContext.hadoopRDD` 方法，它可以指定任意的 `JobConf` ，输入格式(InputFormat)，key 类型，values 类型。你可以跟设置 Hadoop job 一样的方法设置输入源。你还可以在新的 MapReduce 接口(org.apache.hadoop.mapreduce)基础上使用 `SparkContext.newAPIHadoopRDD` (译者注：老的接口是 `SparkContext.newHadoopRDD`)。

- `RDD.saveAsObjectFile` 和 `SparkContext.objectFile` 支持保存一个RDD，保存格式是一个简单的 Java 对象序列化格式。这是一种效率不高的专有格式，如 Avro，它提供了简单的方法来保存任何一个 RDD。

RDD 操作

RDDs 支持 2 种类型的操作：转换*(transformations)* 从已经存在的数据集中创建一个新的数据集；动作*(actions)* 在数据集上进行计算之后返回一个值到驱动程序。例如， `map` 是一个转换操作，它将每一个数据集元素传递给一个函数并且返回一个新的 RDD。另一方面， `reduce` 是一个动作，它使用相同的函数来聚合 RDD 的所有元素，并且将最终的结果返回到驱动程序(不过也有一个并行 `reduceByKey` 能返回一个分布式数据集)。

在 Spark 中，所有的转换(transformations)都是惰性(lazy)的，它们不会马上计算它们的结果。相反的，它们仅仅记录转换操作是应用到哪些基础数据集(例如一个文件)上的。转换仅仅在这个时候计算：当动作(action) 需要一个结果返回给驱动程序的时候。这个设计能够让 Spark 运行得更加高效。例如，我们可以实现：通过 `map` 创建一个新数据集在 `reduce` 中使用，并且仅仅返回 `reduce` 的结果给 driver，而不是整个大的映射过的数据集。

默认情况下，每一个转换过的 RDD 会在每次执行动作(action)的时候重新计算一次。然而，你也可以使用 `persist` (或 `cache`)方法持久化(`persist`)一个 RDD 到内存中。在这个情况下，Spark 会在集群上保存相关的元素，在你下次查询的时候会变得更快。在这里也同样支持持久化 RDD 到磁盘，或在多个节点间复制。

基础

为了说明 RDD 基本知识，考虑下面的简单程序：

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

第一行是定义来自于外部文件的 RDD。这个数据集并没有加载到内存或做其他的操作： `lines` 仅仅是一个指向文件的指针。第二行是定义 `lineLengths` ，它是 `map` 转换(transformation)的结果。同样， `lineLengths` 由于懒惰模式也没有立即计算。最后，我们执行 `reduce` ，它是一个动作(action)。在这个地方，Spark 把计算分成多个任务(task)，并且让它们运行在多个机器上。每台机器都运行自己的 map 部分和本地 reduce 部分。然后仅仅将结果返回给驱动程序。

如果我们想要再次使用 `lineLengths` ，我们可以添加：

```
lineLengths.persist()
```

在 `reduce` 之前，它会导致 `lineLengths` 在第一次计算完成之后保存到内存中。

传递函数到 Spark

Spark 的 API 很大程度上依靠在驱动程序里传递函数到集群上运行。这里有两种推荐的方式：

- 匿名函数 (Anonymous function syntax)，可以在比较短的代码中使用。
- 全局单例对象里的静态方法。例如，你可以定义 `object MyFunctions` 然后传递 `MyFounctions.func1`，像下面这样：

```
object MyFunctions {
  def func1(s: String): String = { ... }
}

myRdd.map(MyFunctions.func1)
```

注意，它可能传递的是一个类实例里的一个方法引用(而不是一个单例对象)，这里必须传送包含方法的整个对象。例如：

```
class MyClass {
  def func1(s: String): String = { ... }
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(func1) }
}
```

这里，如果我们创建了一个 `new MyClass` 对象，并且调用它的 `doStuff`，`map` 里面引用了这个 `MyClass` 实例中的 `func1` 方法，所以这个对象必须传送到集群上。类似写成 `rdd.map(x => this.func1(x))`。

以类似的方式，访问外部对象的字段将会引用整个对象：

```
class MyClass {
  val field = "Hello"
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(x => field + x) }
}
```

相当于写成 `rdd.map(x => this.field + x)`，引用了整个 `this` 对象。为了避免这个问题，最简单的方式是复制 `field` 到一个本地变量而不是从外部访问它：

```
def doStuff(rdd: RDD[String]): RDD[String] = {
  val field_ = this.field
  rdd.map(x => field_ + x)
}
```

使用键值对

虽然很多 Spark 操作工作在包含任意类型对象的 RDDs 上的，但是少数几个特殊操作仅仅在键值(key-value)对 RDDs 上可用。最常见的是分布式 "shuffle" 操作，例如根据一个 key 对一组数据进行分组和聚合。

在 Scala 中，这些操作在包含二元组(Tuple2)(在语言的内建元组中，通过简单的写 (a, b) 创建) 的 RDD 上自动地变成可用的，只要在你的程序中导入 `org.apache.spark.SparkContext._` 来启用 Spark 的隐式转换。在 PairRDDFunctions 的类里键值对操作是可以使用的，如果你导入隐式转换它会自动地包装成元组 RDD。

例如，下面的代码在键值对上使用 `reduceByKey` 操作来统计在一个文件里每一行文本内容出现的次数：

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

我们也可以使用 `counts.sortByKey()` ，例如，将键值对按照字母进行排序，最后 `counts.collect()` 把它们作为一个对象数组带回到驱动程序。

注意：当使用一个自定义对象作为 key 在使用键值对操作的时候，你需要确保自定义 `equals()` 方法和 `hashCode()` 方法是匹配的。更加详细的内容，查看 Object.hashCode() 文档)中的契约概述。

Transformations

下面的表格列了 Spark 支持的一些常用 transformations。详细内容请参阅 RDD API 文档(Scala, Java, Python) 和 PairRDDFunctions 文档(Scala, Java)。

Transformation	Meaning
map(*func*)	返回一个新的分布式数据集，将数据源的每一个元素传递给函数 *func* 映射组成。
filter(*func*)	返回一个新的数据集，从数据源中选中一些元素通过函数 *func* 返回 true。
flatMap(*func*)	类似于 map，但是每个输入项能被映射成多个输出项(所以 *func* 必须返回一个 Seq，而不是单个 item)。
mapPartitions(*func*)	类似于 map，但是分别运行在 RDD 的每个分区上，所以 *func* 的类型必须是 `Iterator<T> => Iterator<U>` 当运行在类型为 T 的 RDD 上。
mapPartitionsWithIndex(*func*)	类似于 mapPartitions，但是 *func* 需要提供一个 integer 值描述索引(index)，所以 *func* 的类型必须是 (Int, Iterator) => Iterator 当运行在类型为 T 的 RDD 上。
sample(withReplacement, fraction, seed)	对数据进行采样。
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or combineByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are also supported through leftOuterJoin and rightOuterJoin.
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable, Iterable) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

Actions

下面的表格列了 Spark 支持的一些常用 actions。详细内容请参阅 RDD API 文档(Scala, Java, Python) 和 PairRDDFunctions 文档(Scala, Java)。

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
	Write the elements of the dataset in a simple format

and Scala)	using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

RDD 持久化

Spark最重要的一个功能是它可以通过各种操作（operations）持久化（或者缓存）一个集合到内存中。当你持久化一个RDD的时候，每一个节点都将参与计算的所有分区数据存储到内存中，并且这些 数据可以被这个集合（以及这个集合衍生的其他集合）的动作（action）重复利用。这个能力使后续的动作速度更快（通常快10倍以上）。对应迭代算法和快速的交互使用来说，缓存是一个关键的工具。

你能通过 `persist()` 或者 `cache()` 方法持久化一个rdd。首先，在action中计算得到rdd；然后，将其保存在每个节点的内存中。Spark的缓存是一个容错的技术-如果RDD的任何一个分区丢失，它 可以通过原有的转换（transformations）操作自动的重复计算并且创建出这个分区。

此外，我们可以利用不同的存储级别存储每一个被持久化的RDD。例如，它允许我们持久化集合到磁盘上、将集合作为序列化的Java对象持久化到内存中、在节点间复制集合或者存储集合到 Tachyon中。我们可以通过传递一个 `StorageLevel` 对象给 `persist()` 方法设置这些存储级别。 `cache()` 方法使用了默认的存储级别— `StorageLevel.MEMORY_ONLY` 。完整的存储级别介绍如下所示：

Storage Level	Meaning
MEMORY_ONLY	将RDD作为非序列化的Java对象存储在jvm中。如果RDD不适合存在内存中，一些分区将不会被缓存，从而在每次需要这些分区时都需重新计算它们。这是系统默认的存储级别。
MEMORY_AND_DISK	将RDD作为非序列化的Java对象存储在jvm中。如果RDD不适合存在内存中，将这些不适合存在内存中的分区存储在磁盘中，每次需要时读出它们。
MEMORY_ONLY_SER	将RDD作为序列化的Java对象存储（每个分区一个byte数组）。这种方式比非序列化方式更节省空间，特别是用到快速的序列化工具时，但是会更耗费cpu资源—密集的读操作。
MEMORY_AND_DISK_SER	和MEMORY_ONLY_SER类似，但不是在每次需要时重复计算这些不适合存储到内存中的分区，而是将这些分区存储到磁盘中。
DISK_ONLY	仅仅将RDD分区存储到磁盘中
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	和上面的存储级别类似，但是复制每个分区到集群的两个节点上面
OFF_HEAP (experimental)	以序列化的格式存储RDD到Tachyon中。相对于MEMORY_ONLY_SER，OFF_HEAP减少了垃圾回收的花费，允许更小的执行者共享内存池。这使其在拥有大量内存的环境下或者多并发应用程序的环境中具有更强的吸引力。

NOTE:在python中，存储的对象都是通过Pickle库序列化了的，所以是否选择序列化等级并不重要。

Spark也会自动持久化一些shuffle操作（如 `reduceByKey` ）中的中间数据，即使用户没有调用 `persist` 方法。这样的好处是避免了在shuffle出错情况下，需要重复计算整个输入。如果用户计划重用 计算过程中产生的RDD，我们仍然推荐用户调用 `persist` 方法。

如何选择存储级别

Spark的多个存储级别意味着在内存利用率和cpu利用效率间的不同权衡。我们推荐通过下面的过程选择一个合适的存储级别：

- 如果你的RDD适合默认的存储级别（MEMORY_ONLY），就选择默认的存储级别。因为这是cpu利用率最高的选项，会使RDD上的操作尽可能的快。
- 如果不适合用默认的级别，选择MEMORY_ONLY_SER。选择一个更快的序列化库提高对象的空间使用率，但是仍能够相当快的访问。

- 除非函数计算RDD的花费较大或者它们需要过滤大量的数据，不要将RDD存储到磁盘上，否则，重复计算一个分区就会和重磁盘上读取数据一样慢。
- 如果你希望更快的错误恢复，可以利用重复(replicated)存储级别。所有的存储级别都可以通过重复计算丢失的数据来支持完整的容错，但是重复的数据能够使你在RDD上继续运行任务，而不需要重复计算丢失的数据。
- 在拥有大量内存的环境中或者多应用程序的环境中，OFF_HEAP具有如下优势：
    - 它运行多个执行者共享Tachyon中相同的内存池
    - 它显著地减少垃圾回收的花费
    - 如果单个的执行者崩溃，缓存的数据不会丢失

删除数据

Spark自动的监控每个节点缓存的使用情况，利用最近最少使用原则删除老旧的数据。如果你想手动的删除RDD，可以使用 `RDD.unpersist()` 方法

共享变量

一般情况下，当一个传递给Spark操作(例如map和reduce)的函数在远程节点上面运行时，Spark操作实际上操作的是这个函数所用变量的一个独立副本。这些变量被复制到每台机器上，并且这些变量在远程机器上 的所有更新都不会传递回驱动程序。通常跨任务的读写变量是低效的，但是，Spark还是为两种常见的使用模式提供了两种有限的共享变量：广播变量（broadcast variable）和累加器（accumulator）

广播变量

广播变量允许程序员缓存一个只读的变量在每台机器上面，而不是每个任务保存一份拷贝。例如，利用广播变量，我们能够以一种更有效率的方式将一个大数据量输入集合的副本分配给每个节点。（Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.）Spark也尝试着利用有效的广播算法去分配广播变量，以减少通信的成本。

一个广播变量可以通过调用 `SparkContext.broadcast(v)` 方法从一个初始变量v中创建。广播变量是v的一个包装变量，它的值可以通过 `value` 方法访问，下面的代码说明了这个过程：

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: spark.Broadcast[Array[Int]] = spark.Broadcast(b5c40191-a864-4c7d-b9bf-d87e
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

广播变量创建以后，我们就能够在集群的任何函数中使用它来代替变量v，这样我们就不需要再次传递变量v到每个节点上。另外，为了保证所有的节点得到广播变量具有相同的值，对象v不能在广播之后被修改。

累加器

顾名思义，累加器是一种只能通过关联操作进行“加”操作的变量，因此它能够高效的应用于并行操作中。它们能够用来实现 `counters` 和 `sums` 。Spark原生支持数值类型的累加器，开发者可以自己添加支持的类型。 如果创建了一个具名的累加器，它可以在spark的UI中显示。这对于理解运行阶段(running stages)的过程有很重要的作用。（注意：这在python中还不被支持）

一个累加器可以通过调用 `SparkContext.accumulator(v)` 方法从一个初始变量v中创建。运行在集群上的任务可以通过 `add` 方法或者使用 `+=` 操作来给它加值。然而，它们无法读取这个值。只有驱动程序可以使用 `value` 方法来读取累加器的值。 如下的代码，展示了如何利用累加器将一个数组里面的所有元素相加：

```
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: spark.Accumulator[Int] = 0
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Int = 10
```

这个例子利用了内置的整数类型累加器。开发者可以利用子类AccumulatorParam创建自己的累加器类型。AccumulatorParam接口有两个方法： `zero` 方法为你的数据类型提供一个“0值”（zero value）； `addInPlace` 方法计算两个值的和。例如，假设我们有一个 `Vector` 类代表数学上的向量，我们能够 如下定义累加器：

```
object VectorAccumulatorParam extends AccumulatorParam[Vector] {
  def zero(initialValue: Vector): Vector = {
    Vector.zeros(initialValue.size)
  }
  def addInPlace(v1: Vector, v2: Vector): Vector = {
    v1 += v2
  }
}
// Then, create an Accumulator of this type:
val vecAccum = sc.accumulator(new Vector(...))(VectorAccumulatorParam)
```

在scala中，Spark支持用更一般的Accumulable接口来累积数据-结果类型和用于累加的元素类型 不一样（例如通过收集的元素建立一个列表）。Spark也支持用 `SparkContext.accumulableCollection` 方法累加一般的scala集合类型。

从这里开始

你能够从spark官方网站查看一些spark运行例子。另外，Spark的example目录包含几个Spark例子，你能够通过如下方式运行Java或者scala例子：

```
./bin/run-example SparkPi
```

为了优化你的项目， configuration和tuning指南提高了最佳实践的信息。保证你保存在内存中的数据是有效的格式是非常重要的事情。为了给部署操作提高帮助，集群模式概述介绍了包含分布式操作和支持集群管理的组件。

最后，完整的API文档可以在后面链接scala,java, python中查看。

Spark Streaming

Spark streaming是Spark核心API的一个扩展，它对实时流式数据的处理具有可扩展性、高吞吐量、可容错性等特点。我们可以从kafka、flume、Twitter、 ZeroMQ、Kinesis等源获取数据，也可以通过由 高阶函数map、reduce、join、window等组成的复杂算法计算出数据。最后，处理后的数据可以推送到文件系统、数据库、实时仪表盘中。事实上，你可以将处理后的数据应用到Spark的机器学习算法、 图处理算法中去。



在内部，它的工作原理如下图所示。Spark Streaming接收实时的输入数据流，然后将这些数据切分为批数据供Spark引擎处理，Spark引擎将数据生成最终的结果数据。



Spark Streaming支持一个高层的抽象，叫做离散流(`discretized stream`)或者 `DStream` ，它代表连续的数据流。DStream既可以利用从Kafka, Flume和Kinesis等源获取的输入数据流创建，也可以 在其他DStream的基础上通过高阶函数获得。在内部，DStream是由一系列RDDs组成。

本指南指导用户开始利用DStream编写Spark Streaming程序。用户能够利用scala、java或者Python来编写Spark Streaming程序。

注意：Spark 1.2已经为Spark Streaming引入了Python API。它的所有DStream transformations和几乎所有的输出操作可以在scala和java接口中使用。然而，它只支持基本的源如文本文件或者套接字上 的文本数据。诸如flume、kafka等外部的源的API会在将来引入。

一个快速的例子

在我们进入如何编写Spark Streaming程序的细节之前，让我们快速地浏览一个简单的例子。在这个例子中，程序从监听TCP套接字的数据服务器获取文本数据，然后计算文本中包含的单词数。做法如下：

首先，我们导入Spark Streaming的相关类以及一些从StreamingContext获得的隐式转换到我们的环境中，为我们所需的其他类（如DStream）提供有用的方法。StreamingContext 是Spark所有流操作的主要入口。然后，我们创建了一个具有两个执行线程以及1秒批间隔时间(即以秒为单位分割数据流)的本地StreamingContext。

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
// Create a local StreamingContext with two working thread and batch interval of 1 second
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

利用这个上下文，我们能够创建一个DStream，它表示从TCP源（主机位localhost，端口为9999）获取的流式数据。

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

这个 `lines` 变量是一个DStream，表示即将从数据服务器获得的流数据。这个DStream的每条记录都代表一行文本。下一步，我们需要将DStream中的每行文本都切分为单词。

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

`flatMap` 是一个一对多的DStream操作，它通过把源DStream的每条记录都生成多条新记录来创建一个新的DStream。在这个例子中，每行文本都被切分成了多个单词，我们把切分 的单词流用 `words` 这个DStream表示。下一步，我们需要计算单词的个数。

```
import org.apache.spark.streaming.StreamingContext._
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print()
```

`words` 这个DStream被mapper(一对一转换操作)成了一个新的DStream，它由（word，1）对组成。然后，我们就可以用这个新的DStream计算每批数据的词频。最后，我们用 `wordCounts.print()` 打印每秒计算的词频。

需要注意的是，当以上这些代码被执行时，Spark Streaming仅仅准备好了它要执行的计算，实际上并没有真正开始执行。在这些转换操作准备好之后，要真正执行计算，需要调用如下的方法

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

完整的例子可以在NetworkWordCount中找到。

如果你已经下载和构建了Spark环境，你就能够用如下的方法运行这个例子。首先，你需要运行Netcat作为数据服务器

```
$ nc -lk 9999
```

然后，在不同的终端，你能够用如下方式运行例子

```
$ ./bin/run-example streaming.NetworkWordCount localhost 9999
```

基本概念

在了解简单的例子的基础上，下面将介绍编写Spark Streaming应用程序必需的一些基本概念。

- 关联
- 初始化StreamingContext
- 离散流
- 输入DStreams
- DStream中的转换
- DStream的输出操作
- 缓存或持久化
- Checkpointing
- 部署应用程序
- 监控应用程序

关联

与Spark类似，Spark Streaming也可以利用maven仓库。编写你自己的Spark Streaming程序，你需要引入下面的依赖到你的SBT或者Maven项目中

```
org.apache.spark
spark-streaming_2.10
1.2
```

为了从Kafka, Flume和Kinesis这些不在Spark核心API中提供的源获取数据，我们需要添加相关的模块 `spark-streaming-xyz_2.10` 到依赖中。例如，一些通用的组件如下表所示：

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

为了获取最新的列表，请访问Apache repository

初始化StreamingContext

为了初始化Spark Streaming程序，一个StreamingContext对象必需被创建，它是Spark Streaming所有流操作的主要入口。一个StreamingContext 对象可以用SparkConf对象创建。

```
import org.apache.spark._
import org.apache.spark.streaming._
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

`appName` 表示你的应用程序显示在集群UI上的名字， `master` 是一个Spark、Mesos、YARN集群URL 或者一个特殊字符串“local[*]”，它表示程序用本地模式运行。当程序运行在集群中时，你并不希望在程序中硬编码 `master` ，而是希望用 `spark-submit` 启动应用程序，并从 `spark-submit` 中得到 `master` 的值。对于本地测试或者单元测试，你可以传递“local”字符串在同一个进程内运行Spark Streaming。需要注意的是，它在内部创建了一个SparkContext对象，你可以通过 `ssc.sparkContext` 访问这个SparkContext对象。

批时间片需要根据你的程序的潜在需求以及集群的可用资源来设定，你可以在性能调优那一节获取详细的信息。

可以利用已经存在的 `SparkContext` 对象创建 `StreamingContext` 对象。

```
import org.apache.spark.streaming._
val sc = ...                // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

当一个上下文（context）定义之后，你必须按照以下几步进行操作

- 定义输入源；
- 准备好流计算指令；
- 利用 `streamingContext.start()` 方法接收和处理数据；
- 处理过程将一直持续，直到 `streamingContext.stop()` 方法被调用。

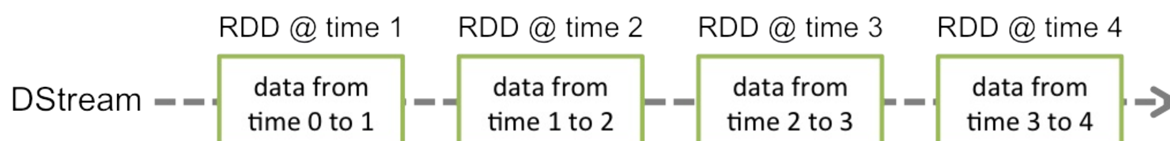
几点需要注意的地方：

- 一旦一个context已经启动，就不能有新的流算子建立或者是添加到context中。
- 一旦一个context已经停止，它就不能再重新启动
- 在JVM中，同一时间只能有一个StreamingContext处于活跃状态
- 在StreamingContext上调用 `stop()` 方法，也会关闭SparkContext对象。如果只想仅关闭StreamingContext对象，设置 `stop()` 的可选参数为false
- 一个SparkContext对象可以重复利用去创建多个StreamingContext对象，前提条件是前

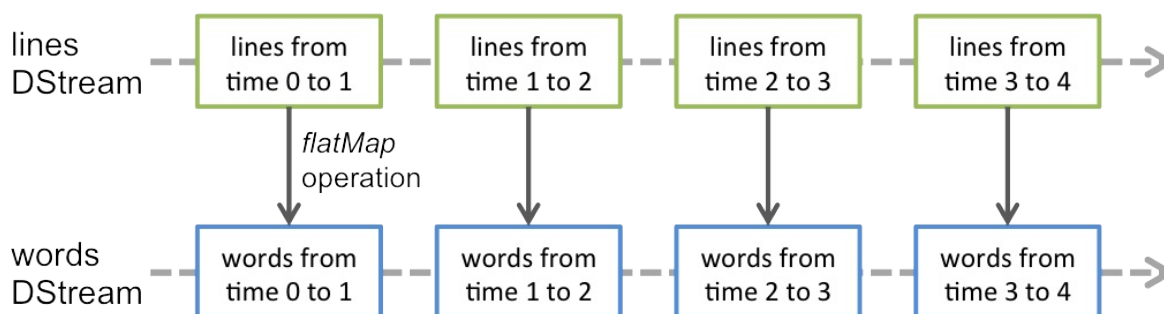
面的StreamingContext在后面StreamingContext创建之前关闭（不关闭SparkContext）。

离散流（DStreams）

离散流或者DStreams是Spark Streaming提供的基本的抽象，它代表一个连续的数据流。它要么是从源中获取的输入流，要么是输入流通过转换算子生成的处理后的数据流。在内部，DStreams由一系列连续的 RDD组成。DStreams中的每个RDD都包含确定时间间隔内的数据，如下图所示：



任何对DStreams的操作都转换成了对DStreams隐含的RDD的操作。在前面的例子中， `flatMap` 操作应用于 `lines` 这个DStreams的每个RDD，生成 `words` 这个DStreams的RDD。过程如下图所示：



通过Spark引擎计算这些隐含RDD的转换算子。DStreams操作隐藏了大部分的细节，并且为了更便捷，为开发者提供了更高层的API。下面几节将具体讨论这些操作的细节。

输入DStreams和receivers

输入DStreams表示从数据源获取输入数据流的DStreams。在快速例子中， `lines` 表示输入DStream，它代表从netcat服务器获取的数据流。每一个输入流DStream 和一个 `Receiver` 对象相关联，这个 `Receiver` 从源中获取数据，并将数据存入内存中用于处理。

输入DStreams表示从数据源获取的原始数据流。Spark Streaming拥有两类数据源

- 基本源（Basic sources）：这些源在StreamingContext API中直接可用。例如文件系统、套接字连接、Akka的actor等。
- 高级源（Advanced sources）：这些源包括Kafka,Flume,Kinesis,Twitter等等。它们需要通过额外的类来使用。我们在关联那一节讨论了类依赖。

需要注意的是，如果你想在一个流应用中并行地创建多个输入DStream来接收多个数据流，你能够创建多个输入流（这将在性能调优那一节介绍）。它将创建多个Receiver同时接收多个数据流。但是， `receiver` 作为一个长期运行的任务运行在Spark worker或executor中。因此，它占有一个核，这个核是分配给Spark Streaming应用程序的所有 核中的一个（it occupies one of the cores allocated to the Spark Streaming application）。所以，为Spark Streaming应用程序分配足够的核（如果是本地运行，那么是线程） 用以处理接收的数据并且运行 `receiver` 是非常重要的。

几点需要注意的地方：

- 如果分配给应用程序的核的数量少于或者等于输入DStreams或者receivers的数量，系统只能够接收数据而不能处理它们。
- 当运行在本地，如果你的master URL被设置成了“local”，这样就只有一个核运行任务。这对程序来说是不足的，因为作为 `receiver` 的输入DStream将会占用这个核，这样就没有剩余的核来处理数据了。

基本源

我们已经在快速例子中看到， `ssc.socketTextStream(...)` 方法用来把从TCP套接字获取的文本数据创建成DStream。除了套接字，StreamingContext API也支持把文件 以及Akka actors作为输入源创建DStream。

- 文件流（File Streams）：从任何与HDFS API兼容的文件系统中读取数据，一个DStream可以通过如下方式创建

```
streamingContext.fileStream[keyClass, valueClass, inputFormatClass](dataDirectory)
```

Spark Streaming将会监控 `dataDirectory` 目录，并且处理目录下生成的任何文件（嵌套目录不被支持）。需要注意一下三点：

```
1 所有文件必须具有相同的数据格式
2 所有文件必须在`dataDirectory`目录下创建，文件是自动的移动和重命名到数据目录下
3 一旦移动，文件必须被修改。所以如果文件被持续的附加数据，新的数据不会被读取。
```

对于简单的文本文件，有一个更简单的方法 `streamingContext.textFileStream(dataDirectory)` 可以被调用。文件流不需要运行一个receiver，所以不需要分配核。

在Spark1.2中， `fileStream` 在Python API中不可用，只有 `textFileStream` 可用。

- 基于自定义actor的流：DStream可以调用 `streamingContext.actorStream(actorProps, actor-name)` 方法从Akka actors获取的数据流来创建。具体的信息见自定义receiver指南 `actorStream` 在Python API中不可用。
- RDD队列作为数据流：为了用测试数据测试Spark Streaming应用程序，人们也可以调用 `streamingContext.queueStream(queueOfRDDs)` 方法基于RDD队列创建DStreams。每个push到队列的RDD都被 当做DStream的批数据，像流一样处理。

关于从套接字、文件和actor中获取流的更多细节，请看StreamingContext和JavaStreamingContext

高级源

这类源需要非Spark库接口，并且它们中的部分还需要复杂的依赖（例如kafka和flume）。为了减少依赖的版本冲突问题，从这些源创建DStream的功能已经被移到了独立的库中，你能在关联查看 细节。例如，如果你想用来自推特的流数据创建DStream，你需要按照如下步骤操作：

- 关联：添加 `spark-streaming-twitter_2.10` 到SBT或maven项目的依赖中
- 编写：导入 `TwitterUtils` 类，用 `TwitterUtils.createStream` 方法创建DStream,如下所示

```scala
import org.apache.spark.streaming.twitter._
TwitterUtils.createStream(ssc)
```

- 部署：将编写的程序以及其所有的依赖（包括spark-streaming-twitter_2.10的依赖以及它的传递依赖）打为jar包，然后部署。这在部署章节将会作更进一步的介绍。

需要注意的是，这些高级的源在 `spark-shell` 中不能被使用，因此基于这些源的应用程序无法在shell中测试。

下面将介绍部分的高级源：

- Twitter：Spark Streaming利用 `Twitter4j 3.0.3` 获取公共的推文流，这些推文通过推特流API获得。认证信息可以通过Twitter4J库支持的 任何方法提供。你既能够得到公共流，也能够得到基于关键字过滤后的流。你可以查看API文档（scala和java） 和例子（TwitterPopularTags和TwitterAlgebirdCMS）
- Flume：Spark Streaming 1.2能够从flume 1.4.0中获取数据，可以查看flume集成指南了解详细信息
- Kafka：Spark Streaming 1.2能够从kafka 0.8.0中获取数据，可以查看kafka集成指南了解详细信息
- Kinesis：查看Kinesis集成指南了解详细信息

自定义源

在Spark 1.2中，这些源不被Python API支持。 输入DStream也可以通过自定义源创建，你需要做的是实现用户自定义的 `receiver` ，这个 `receiver` 可以从自定义源接收数据以及将数据推到Spark中。通过自定义receiver指南了解详细信息

Receiver可靠性

基于可靠性有两类数据源。源(如kafka、flume)允许。如果从这些可靠的源获取数据的系统能够正确的应答所接收的数据，它就能够确保在任何情况下不丢失数据。这样，就有两种类型的receiver：

- Reliable Receiver：一个可靠的receiver正确的应答一个可靠的源，数据已经收到并且被正确地复制到了Spark中。
- Unreliable Receiver ：这些receivers不支持应答。即使对于一个可靠的源，开发者可能实现一个非可靠的receiver，这个receiver不会正确应答。

怎样编写可靠的Receiver的细节在自定义receiver中有详细介绍。

DStream中的转换（transformation）

和RDD类似，transformation允许从输入DStream来的数据被修改。DStreams支持很多在RDD中可用的transformation算子。一些常用的算子如下所示：

Transformation	Meaning
map(func)	利用函数 `func` 处理原DStream的每个元素，返回一个新的DStream
flatMap(func)	与map相似，但是每个输入项可用被映射为0个或者多个输出项
filter(func)	返回一个新的DStream，它仅仅包含源DStream中满足函数func的项
repartition(numPartitions)	通过创建更多或者更少的partition改变这个DStream的并行级别(level of parallelism)
union(otherStream)	返回一个新的DStream,它包含源DStream和otherStream的联合元素
count()	通过计算源DStream中每个RDD的元素数量，返回一个包含单元素(single-element)RDDs的新DStream
reduce(func)	利用函数func聚集源DStream中每个RDD的元素，返回一个包含单元素(single-element)RDDs的新DStream。函数应该是相关联的，以使计算可以并行化
countByValue()	这个算子应用于元素类型为K的DStream上，返回一个（K,long）对的新DStream，每个键的值是在原DStream的每个RDD中的频率。
reduceByKey(func, [numTasks])	当在一个由(K,V)对组成的DStream上调用这个算子，返回一个新的由(K,V)对组成的DStream，每一个key的值均由给定的reduce函数聚集起来。注意：在默认情况下，这个算子利用了Spark默认的并发任务数去分组。你可以用 `numTasks` 参数设置不同的任务数
join(otherStream, [numTasks])	当应用于两个DStream（一个包含（K,V）对,一个包含(K,W)对），返回一个包含(K, (V, W))对的新DStream
cogroup(otherStream, [numTasks])	当应用于两个DStream（一个包含（K,V）对,一个包含(K,W)对），返回一个包含(K, Seq[V], Seq[W])的元组
transform(func)	通过对源DStream的每个RDD应用RDD-to-RDD函数，创建一个新的DStream。这个可以在DStream中的任何RDD操作中使用
updateStateByKey(func)	利用给定的函数更新DStream的状态，返回一个新"state"的DStream。

最后两个transformation算子需要重点介绍一下：

UpdateStateByKey操作

updateStateByKey操作允许不断用新信息更新它的同时保持任意状态。你需要通过两步来使用它

- 定义状态-状态可以是任何的数据类型
- 定义状态更新函数-怎样利用更新前的状态和从输入流里面获取的新值更新状态

让我们举个例子说明。在例子中，你想保持一个文本数据流中每个单词的运行次数，运行次数用一个state表示，它的类型是整数

```
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
    val newCount = ...  // add the new values with the previous running count to get the
    Some(newCount)
}
```

这个函数被用到了DStream包含的单词上

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
// Create a local StreamingContext with two working thread and batch interval of 1 second
val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
// Split each line into words
val words = lines.flatMap(_.split(" "))
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

更新函数将会被每个单词调用，`newValues` 拥有一系列的1（从 (词, 1)对而来），runningCount拥有之前的次数。要看完整的代码，见例子

Transform操作

`transform` 操作（以及它的变化形式如 `transformWith` ）允许在DStream运行任何RDD-to-RDD函数。它能够被用来应用任何没在DStream API中提供的RDD操作（It can be used to apply any RDD operation that is not exposed in the DStream API）。 例如，连接数据流中的

每个批（batch）和另外一个数据集的功能并没有在DStream API中提供，然而你可以简单的利用 `transform` 方法做到。如果你想通过连接带有预先计算的垃圾邮件信息的输入数据流 来清理实时数据，然后过了它们，你可以按如下方法来做：

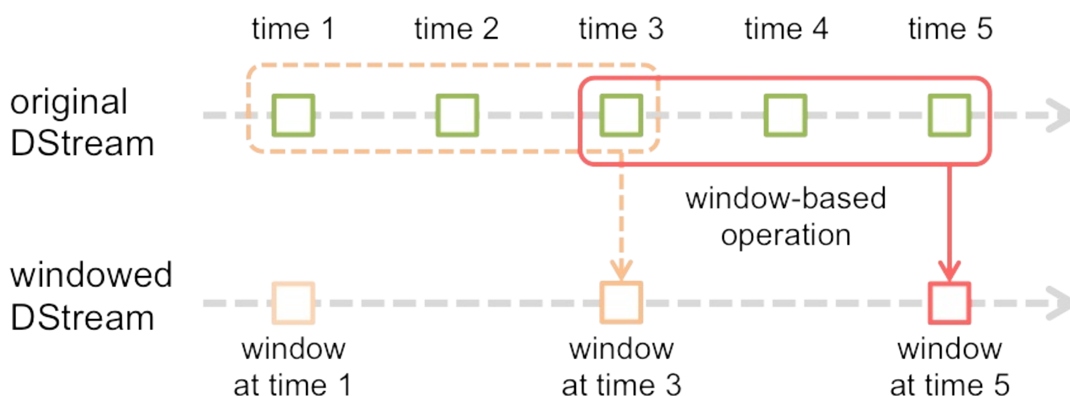
```
val spamInfoRDD = ssc.sparkContext.newAPIHadoopRDD(...) // RDD containing spam informatio

val cleanedDStream = wordCounts.transform(rdd => {
  rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data
  ...
})
```

事实上，你也可以在 `transform` 方法中用机器学习和图计算算法

窗口(window)操作

Spark Streaming也支持窗口计算，它允许你在一个滑动窗口数据上应用transformation算子。下图阐明了这个滑动窗口。



如上图显示，窗口在源DStream上滑动，合并和操作落入窗内的源RDDs，产生窗口化的DStream的RDDs。在这个具体的例子中，程序在三个时间单元的数据上进行窗口操作，并且每两个时间单元滑动一次。 这说明，任何一个窗口操作都需要指定两个参数：

- 窗口长度：窗口的持续时间
- 滑动的时间间隔：窗口操作执行的时间间隔

这两个参数必须是源DStream的批时间间隔的倍数。

下面举例说明窗口操作。例如，你想扩展前面的例子用来计算过去30秒的词频，间隔时间是10秒。为了达到这个目的，我们必须在过去30秒的 `pairs` DStream上应用 `reduceByKey` 操作。用方法 `reduceByKeyAndWindow` 实现。

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30)
```

一些常用的窗口操作如下所示，这些操作都需要用到上文提到的两个参数：窗口长度和滑动的时间间隔

Transformation	Meaning
window(windowLength, slideInterval)	基于源DStream产生的窗口化的批数据计算一个新的DStream
countByWindow(windowLength, slideInterval)	返回流中元素的一个滑动窗口数
reduceByWindow(func, windowLength, slideInterval)	返回一个单元素流。利用函数func聚集滑动时间间隔的流的元素创建这个单元素流。函数必须是相关联的以使计算能够正确的并行计算。
reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])	应用到一个(K,V)对组成的DStream上，返回一个由(K,V)对组成的新的DStream。每一个key的值均由给定的reduce函数聚集起来。注意：在默认情况下，这个算子利用了Spark默认的并发任务数去分组。你可以用 `numTasks` 参数设置不同的任务数
reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])	A more efficient version of the above reduceByKeyAndWindow() where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enter the sliding window, and "inverse reducing" the old data that leave the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable to only "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter invFunc. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument.
countByValueAndWindow(windowLength, slideInterval, [numTasks])	应用到一个(K,V)对组成的DStream上，返回一个由(K,V)对组成的新的DStream。每个key的值都是它们在滑动窗口中出现的频率。

DStreams上的输出操作

输出操作允许DStream的操作推到如数据库、文件系统等外部系统中。因为输出操作实际上是允许外部系统消费转换后的数据，它们触发的实际操作是DStream转换。目前，定义了下面几种输出操作：

Output Operation	Meaning
print()	在DStream的每个批数据中打印前10条元素，这个操作在开发和调试中都非常有用。在Python API中调用 `pprint()` 。
saveAsObjectFiles(prefix, [suffix])	保存DStream的内容为一个序列化的文件 `SequenceFile` 。每一个批间隔的文件的文件名基于 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"，在Python API中不可用。
saveAsTextFiles(prefix, [suffix])	保存DStream的内容为一个文本文件。每一个批间隔的文件的文件名基于 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"
saveAsHadoopFiles(prefix, [suffix])	保存DStream的内容为一个hadoop文件。每一个批间隔的文件的文件名基于 `prefix` 和 `suffix` 生成。"prefix-TIME_IN_MS[.suffix]"，在Python API中不可用。
foreachRDD(func)	在从流中生成的每个RDD上应用函数 `func` 的最通用的输出操作。这个函数应该推送每个RDD的数据到外部系统，例如保存RDD到文件或者通过网络写到数据库中。需要注意的是， `func` 函数在驱动程序中执行，并且通常都有RDD action在里面推动RDD流的计算。

利用foreachRDD的设计模式

dstream.foreachRDD是一个强大的原语，发送数据到外部系统中。然而，明白怎样正确地、有效地用这个原语是非常重要的。下面几点介绍了如何避免一般错误。

- 经常写数据到外部系统需要建一个连接对象（例如到远程服务器的TCP连接），用它发送数据到远程系统。为了达到这个目的，开发人员可能不经意的在Spark驱动中创建一个连接对象，但是在Spark worker中 尝试调用这个连接对象保存记录到RDD中，如下：

```
dstream.foreachRDD(rdd => {
    val connection = createNewConnection()  // executed at the driver
    rdd.foreach(record => {
        connection.send(record) // executed at the worker
    })
})
```

这是不正确的，因为这需要先序列化连接对象，然后将它从driver发送到worker中。这样的连接对象在机器之间不能传送。它可能表现为序列化错误（连接对象不可序列化）或者初始化错误（连接对象应该 在worker中初始化）等等。正确的解决办法是在worker中创建连接对象。

- 然而，这会造成另外一个常见的错误-为每一个记录创建了一个连接对象。例如：

```
dstream.foreachRDD(rdd => {
    rdd.foreach(record => {
        val connection = createNewConnection()
        connection.send(record)
        connection.close()
    })
})
```

通常，创建一个连接对象有资源和时间的开支。因此，为每个记录创建和销毁连接对象会导致非常高的开支，明显的减少系统的整体吞吐量。一个更好的解决办法是利用 `rdd.foreachPartition` 方法。 为RDD的partition创建一个连接对象，用这个两件对象发送partition中的所有记录。

```
dstream.foreachRDD(rdd => {
     rdd.foreachPartition(partitionOfRecords => {
         val connection = createNewConnection()
         partitionOfRecords.foreach(record => connection.send(record))
         connection.close()
     })
 })
```

这就将连接对象的创建开销分摊到了partition的所有记录上了。

- 最后，可以通过在多个RDD或者批数据间重用连接对象做更进一步的优化。开发者可以保有一个静态的连接对象池，重复使用池中的对象将多批次的RDD推送到外部系统，以进一步节省开支。

```
dstream.foreachRDD(rdd => {
    rdd.foreachPartition(partitionOfRecords => {
        // ConnectionPool is a static, lazily initialized pool of connections
        val connection = ConnectionPool.getConnection()
        partitionOfRecords.foreach(record => connection.send(record))
        ConnectionPool.returnConnection(connection)  // return to the pool for future r
    })
})
```

需要注意的是，池中的连接对象应该根据需要延迟创建，并且在空闲一段时间后自动超时。这样就获取了最有效的方式发生数据到外部系统。

其它需要注意的地方：

- 输出操作通过懒执行的方式操作DStreams，正如RDD action通过懒执行的方式操作RDD。具体地看，RDD actions和DStreams输出操作接收数据的处理。因此，如果你的应用程序没有任何输出操作或者 用于输出操作 `dstream.foreachRDD()` ，但是没有任何RDD action操作在 `dstream.foreachRDD()` 里面，那么什么也不会执行。系统仅仅会接收输入，然后丢弃它们。
- 默认情况下，DStreams输出操作是分时执行的，它们按照应用程序的定义顺序按序执行。

缓存或持久化

和RDD相似，DStreams也允许开发者持久化流数据到内存中。在DStream上使用 `persist()` 方法可以自动地持久化DStream中的RDD到内存中。如果DStream中的数据需要计算多次，这是非常有用的。像 `reduceByWindow` 和 `reduceByKeyAndWindow` 这种窗口操作、 `updateStateByKey` 这种基于状态的操作，持久化是默认的，不需要开发者调用 `persist()` 方法。

例如通过网络（如kafka，flume等）获取的输入数据流，默认的持久化策略是复制数据到两个不同的节点以容错。

注意，与RDD不同的是，DStreams默认持久化级别是存储序列化数据到内存中，这将在性能调优章节介绍。更多的信息请看rdd持久化

Checkpointing

一个流应用程序必须全天候运行，所有必须能够解决应用程序逻辑无关的故障（如系统错误，JVM崩溃等）。为了使这成为可能，Spark Streaming需要checkpoint足够的信息到容错存储系统中， 以使系统从故障中恢复。

- Metadata checkpointing：保存流计算的定义信息到容错存储系统如HDFS中。这用来恢复应用程序中运行worker的节点的故障。元数据包括
    - Configuration ：创建Spark Streaming应用程序的配置信息
    - DStream operations ：定义Streaming应用程序的操作集合
    - Incomplete batches：操作存在队列中的未完成的批
- Data checkpointing ：保存生成的RDD到可靠的存储系统中，这在有状态transformation（如结合跨多个批次的数据）中是必须的。在这样一个transformation中，生成的RDD依赖于之前 批的RDD，随着时间的推移，这个依赖链的长度会持续增长。在恢复的过程中，为了避免这种无限增长。有状态的transformation的中间RDD将会定时地存储到可靠存储系统中，以截断这个依赖链。

元数据checkpoint主要是为了从driver故障中恢复数据。如果transformation操作被用到了，数据checkpoint即使在简单的操作中都是必须的。

何时checkpoint

应用程序在下面两种情况下必须开启checkpoint

- 使用有状态的transformation。如果在应用程序中用到了 `updateStateByKey` 或者 `reduceByKeyAndWindow` ，checkpoint目录必需提供用以定期checkpoint RDD。
- 从运行应用程序的driver的故障中恢复过来。使用元数据checkpoint恢复处理信息。

注意，没有前述的有状态的transformation的简单流应用程序在运行时可以不开启checkpoint。在这种情况下，从driver故障的恢复将是部分恢复（接收到了但是还没有处理的数据将会丢失）。 这通常是可以接受的，许多运行的Spark Streaming应用程序都是这种方式。

怎样配置Checkpointing

在容错、可靠的文件系统（HDFS、s3等）中设置一个目录用于保存checkpoint信息。着可以通过 `streamingContext.checkpoint(checkpointDirectory)` 方法来做。这运行你用之前介绍的 有状态transformation。另外，如果你想从driver故障中恢复，你应该以下面的方式重写你的Streaming应用程序。

- 当应用程序是第一次启动，新建一个StreamingContext，启动所有Stream，然后调用 `start()` 方法
- 当应用程序因为故障重新启动，它将会从checkpoint目录checkpoint数据重新创建StreamingContext

```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // new context
    val lines = ssc.socketTextStream(...) // create DStreams
    ...
    ssc.checkpoint(checkpointDirectory)   // set checkpoint directory
    ssc
}

// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext _

// Do additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted
context. ...

// Start the context
context.start()
context.awaitTermination()
```

如果 `checkpointDirectory` 存在，上下文将会利用checkpoint数据重新创建。如果这个目录不存在，将会调用 `functionToCreateContext` 函数创建一个新的上下文，建立DStreams。 请看RecoverableNetworkWordCount例子。

除了使用 `getOrCreate` ，开发者必须保证在故障发生时，driver处理自动重启。只能通过部署运行应用程序的基础设施来达到该目的。在部署章节将有更进一步的讨论。

注意，RDD的checkpointing有存储成本。这会导致批数据（包含的RDD被checkpoint）的处理时间增加。因此，需要小心的设置批处理的时间间隔。在最小的批容量(包含1秒的数据)情况下，checkpoint每批数据会显著的减少 操作的吞吐量。相反，checkpointing太少会导致谱系以及任务大小增大，这会产生有害的影响。因为有状态的transformation需要RDD checkpoint。默认的间隔时间是批间隔时间的倍数，最少10秒。它可以通过 `dstream.checkpoint` 来设置。典型的情况下，设置checkpoint间隔是DStream的滑动间隔的5-10大小是一个好的尝试。

部署应用程序

Requirements

运行一个Spark Streaming应用程序，有下面一些步骤

- 有管理器的集群-这是任何Spark应用程序都需要的需求，详见部署指南
- 将应用程序打为jar包-你必须编译你的应用程序为jar包。如果你用spark-submit启动应用程序，你不需要将Spark和Spark Streaming打包进这个jar包。 如果你的应用程序用到了高级源（如kafka，flume），你需要将它们关联的外部artifact以及它们的依赖打包进需要部署的应用程序jar包中。例如，一个应用程序用到了 `TwitterUtils` ，那么就需要将 `spark-streaming-twitter_2.10` 以及它的所有依赖打包到应用程序jar中。
- 为executors配置足够的内存-因为接收的数据必须存储在内存中，executors必须配置足够的内存用来保存接收的数据。注意，如果你正在做10分钟的窗口操作，系统的内存要至少能保存10分钟的数据。所以， 应用程序的内存需求依赖于使用 它的操作。
- 配置checkpointing-如果stream应用程序需要checkpointing，然后一个与Hadoop API兼容的容错存储目录必须配置为检查点的目录，流应用程序将checkpoint信息写入该目录用于错误恢复。
- 配置应用程序driver的自动重启-为了自动从driver故障中恢复，运行流应用程序的部署设施必须能监控driver进程，如果失败了能够重启它。不同的集群管理器，有不同的工具得到该功能
  - Spark Standalone：一个Spark应用程序driver可以提交到Spark独立集群运行，也就是说driver运行在一个worker节点上。进一步来看，独立的集群管理器能够被指示用来监控driver，并且在driver失败（或者是由于非零的退出代码如exit(1)， 或者由于运行driver的节点的故障）的情况下重启driver。
  - YARN：YARN为自动重启应用程序提供了类似的机制。
  - Mesos： Mesos可以用Marathon提供该功能
- 配置write ahead logs-在Spark 1.2中，为了获得极强的容错保证，我们引入了一个新的实验性的特性-预写日志（write ahead logs）。如果该特性开启，从receiver获取的所有数据会将预写日志写入配置的checkpoint目录。 这可以防止driver故障丢失数据，从而保证零数据丢失。这个功能可以通过设置配置参数 `spark.streaming.receiver.writeAheadLogs.enable` 为true来开启。然而，这些较强的语义可能以receiver的接收吞吐量为代价。这可以通过 并行运行多个receiver增加吞吐量来解决。另外，当预写日志开启时，Spark中的复制数据的功能推荐不用，因为该日志已经存储在了一个副本在存储系统中。可以通过设置输入DStream的存储级别为 `StorageLevel.MEMORY_AND_DISK_SER` 获得该功能。

升级应用程序代码

如果运行的Spark Streaming应用程序需要升级，有两种可能的方法

- 启动升级的应用程序，使其与未升级的应用程序并行运行。一旦新的程序（与就程序接收相同的数据）已经准备就绪，旧的应用程序就可以关闭。这种方法支持将数据发送到两个不同的目的地（新程序一个，旧程序一个）
- 首先，平滑的关闭（`StreamingContext.stop(...)`或`JavaStreamingContext.stop(...)`）现有的应用程序。在关闭之前，要保证已经接收的数据完全处理完。然后，就可以启动升级的应用程序，升级 的应用程序会接着旧应用程序的点开始处理。这种方法仅支持具有源端缓存功能的输入源（如flume，kafka），这是因为当旧的应用程序已经关闭，升级的应用程序还没有启动的时候，数据需要被缓存。

监控应用程序

除了Spark的监控功能，Spark Streaming增加了一些专有的功能。应用StreamingContext的时候，Spark web UI 显示添加的 `Streaming` 菜单，用以显示运行的receivers（receivers是否是存活状态、接收的记录数、receiver错误等）和完成的批的统计信息（批处理时间、队列等待等待）。这可以用来监控 流应用程序的处理过程。

在WEB UI中的 `Processing Time` 和 `Scheduling Delay` 两个度量指标是非常重要的。第一个指标表示批数据处理的时间，第二个指标表示前面的批处理完毕之后，当前批在队列中的等待时间。如果 批处理时间比批间隔时间持续更长或者队列等待时间持续增加，这就预示系统无法以批数据产生的速度处理这些数据，整个处理过程滞后了。在这种情况下，考虑减少批处理时间。

Spark Streaming程序的处理过程也可以通过StreamingListener接口来监控，这 个接口允许你获得receiver状态和处理时间。注意，这个接口是开发者API，它有可能在未来提供更多的信息。

性能调优

集群中的Spark Streaming应用程序获得最好的性能需要一些调整。这章将介绍几个参数和配置，提高Spark Streaming应用程序的性能。你需要考虑两件事情：

- 高效地利用集群资源减少批数据的处理时间
- 设置正确的批容量（size），使数据的处理速度能够赶上数据的接收速度

减少批数据的执行时间

在Spark中有几个优化可以减少批处理的时间。这些可以在优化指南中作了讨论。这节重点讨论几个重要的。

数据接收的并行水平

通过网络(如kafka，flume，socket等)接收数据需要这些数据反序列化并被保存到Spark中。如果数据接收成为系统的瓶颈，就要考虑并行地接收数据。注意，每个输入DStream创建一个 `receiver` （运行在worker机器上） 接收单个数据流。创建多个输入DStream并配置它们可以从源中接收不同分区的数据流，从而实现多数据流接收。例如，接收两个topic数据的单个输入DStream可以被切分为两个kafka输入流，每个接收一个topic。这将 在两个worker上运行两个 `receiver` ，因此允许数据并行接收，提高整体的吞吐量。多个DStream可以被合并生成单个DStream，这样运用在单个输入DStream的transformation操作可以运用在合并的DStream上。

```
val numStreams = 5
val kafkaStreams = (1 to numStreams).map { i => KafkaUtils.createStream(...) }
val unifiedStream = streamingContext.union(kafkaStreams)
unifiedStream.print()
```

另外一个需要考虑的参数是 `receiver` 的阻塞时间。对于大部分的 `receiver` ，在存入Spark内存之前，接收的数据都被合并成了一个大数据块。每批数据中块的个数决定了任务的个数。这些任务是用类 似map的transformation操作接收的数据。阻塞间隔由配置参数 `spark.streaming.blockInterval` 决定，默认的值是200毫秒。

多输入流或者多 `receiver` 的可选的方法是明确地重新分配输入数据流（利用 `inputStream.repartition(<number of partitions>)` ），在进一步操作之前，通过集群的机器数分配接收的批数据。

数据处理的并行水平

如果运行在计算stage上的并发任务数不足够大，就不会充分利用集群的资源。例如，对于分布式reduce操作如 `reduceByKey` 和 `reduceByKeyAndWindow` ，默认的并发任务数通过配置属性来确定（configuration.html#spark-properties） `spark.default.parallelism` 。你可以通过参数（ `PairDStreamFunctions` (api/scala/index.html#org.apache.spark.streaming.dstream.PairDStreamFunctions)）传递并行度，或者设置参数 `spark.default.parallelism` 修改默认值。

数据序列化

数据序列化的总开销是平常大的，特别是当sub-second级的批数据被接收时。下面有两个相关点：

- Spark中RDD数据的序列化。关于数据序列化请参照Spark优化指南。注意，与Spark不同的是，默认的RDD会被持久化为序列化的字节数组，以减少与垃圾回收相关的暂停。
- 输入数据的序列化。从外部获取数据存到Spark中，获取的byte数据需要从byte反序列化，然后再按照Spark的序列化格式重新序列化到Spark中。因此，输入数据的反序列化花费可能是一个瓶颈。

任务的启动开支

每秒钟启动的任务数是非常大的（50或者更多）。发送任务到slave的花费明显，这使请求很难获得亚秒（sub-second）级别的反应。通过下面的改变可以减小开支

- 任务序列化。运行kyro序列化任何可以减小任务的大小，从而减小任务发送到slave的时间。
- 执行模式。在Standalone模式下或者粗粒度的Mesos模式下运行Spark可以在比细粒度Mesos模式下运行Spark获得更短的任务启动时间。可以在在Mesos下运行Spark中获取更多信息。

These changes may reduce batch processing time by 100s of milliseconds, thus allowing sub-second batch size to be viable.

设置正确的批容量

为了Spark Streaming应用程序能够在集群中稳定运行，系统应该能够以足够的速度处理接收的数据（即处理速度应该大于或等于接收数据的速度）。这可以通过流的网络UI观察得到。批处理时间应该小于批间隔时间。

根据流计算的性质，批间隔时间可能显著的影响数据处理速率，这个速率可以通过应用程序维持。可以考虑 `WordCountNetwork` 这个例子，对于一个特定的数据处理速率，系统可能可以每2秒打印一次单词计数 （批间隔时间为2秒），但无法每500毫秒打印一次单词计数。所以，为了在生产环境中维持期望的数据处理速率，就应该设置合适的批间隔时间(即批数据的容量)。

找出正确的批容量的一个好的办法是用一个保守的批间隔时间（5-10,秒）和低数据速率来测试你的应用程序。为了验证你的系统是否能满足数据处理速率，你可以通过检查端到端的延迟值来判断（可以在 Spark驱动程序的log4j日志中查看"Total delay"或者利用StreamingListener接口）。如果延迟维持稳定，那么系统是稳定的。如果延迟持续增长，那么系统无法跟上数据处理速率，是不稳定的。 你能够尝试着增加数据处理速率或者减少批容量来作进一步的测试。注意，因为瞬间的数据处理速度增加导致延迟瞬间的增长可能是正常的，只要延迟能重新回到了低值（小于批容量）。

内存调优

调整内存的使用以及Spark应用程序的垃圾回收行为已经在Spark优化指南中详细介绍。在这一节，我们重点介绍几个强烈推荐的自定义选项，它们可以 减少Spark Streaming应用程序垃圾回收的相关暂停，获得更稳定的批处理时间。

- Default persistence level of DStreams：和RDDs不同的是，默认的持久化级别是序列化数据到内存中（DStream是 `StorageLevel.MEMORY_ONLY_SER` ，RDD是 `StorageLevel.MEMORY_ONLY` ）。 即使保存数据为序列化形态会增加序列化/反序列化的开销，但是可以明显的减少垃圾回收的暂停。
- Clearing persistent RDDs：默认情况下，通过Spark内置策略（LUR），Spark Streaming生成的持久化RDD将会从内存中清理掉。如果spark.cleaner.ttl已经设置了，比这个时间存在更老的持久化 RDD将会被定时的清理掉。正如前面提到的那样，这个值需要根据Spark Streaming应用程序的操作小心设置。然而，可以设置配置选项 `spark.streaming.unpersist` 为true来更智能的去持久化（unpersist）RDD。这个 配置使系统找出那些不需要经常保有的RDD，然后去持久化它们。这可以减少Spark RDD的内存使用，也可能改善垃圾回收的行为。
- Concurrent garbage collector：使用并发的标记-清除垃圾回收可以进一步减少垃圾回收的暂停时间。尽管并发的垃圾回收会减少系统的整体吞吐量，但是仍然推荐使用它以获得更稳定的批处理时间。

容错语义

这一节，我们将讨论在节点错误事件时Spark Streaming的行为。为了理解这些，让我们先记住一些Spark RDD的基本容错语义。

- 一个RDD是不可变的、确定可重复计算的、分布式数据集。每个RDD记住一个确定性操作的谱系(lineage)，这个谱系用在容错的输入数据集上来创建该RDD。
- 如果任何一个RDD的分区因为节点故障而丢失，这个分区可以通过操作谱系从源容错的数据集中重新计算得到。
- 假定所有的RDD transformations是确定的，那么最终转换的数据是一样的，不论Spark机器中发生何种错误。

Spark运行在像HDFS或S3等容错系统的数据上。因此，任何从容错数据而来的RDD都是容错的。然而，这不是在Spark Streaming的情况下，因为Spark Streaming的数据大部分情况下是从 网络中得到的。为了获得生成的RDD相同的容错属性，接收的数据需要重复保存在worker node的多个Spark executor上（默认的复制因子是2），这导致了当出现错误事件时，有两类数据需要被恢复

- Data received and replicated ：在单个worker节点的故障中，这个数据会幸存下来，因为有另外一个节点保存有这个数据的副本。
- Data received but buffered for replication：因为没有重复保存，所以为了恢复数据，唯一的办法是从源中重新读取数据。

有两种错误我们需要关心

- worker节点故障：任何运行executor的worker节点都有可能出故障，那样在这个节点中的所有内存数据都会丢失。如果有任何receiver运行在错误节点，它们的缓存数据将会丢失
- Driver节点故障：如果运行Spark Streaming应用程序的Driver节点出现故障，很明显SparkContext将会丢失，所有执行在其上的executors也会丢失。

作为输入源的文件语义（Semantics with files as input source）

如果所有的输入数据都存在于一个容错的文件系统如HDFS，Spark Streaming总可以从任何错误中恢复并且执行所有数据。这给出了一个恰好一次(exactly-once)语义，即无论发生什么故障， 所有的数据都将会恰好处理一次。

基于receiver的输入源语义

对于基于receiver的输入源，容错的语义既依赖于故障的情形也依赖于receiver的类型。正如之前讨论的，有两种类型的receiver

- Reliable Receiver：这些receivers只有在确保数据复制之后才会告知可靠源。如果这样一个receiver失败了，缓冲（非复制）数据不会被源所承认。如果receiver重启，源会重发数 据，因此不会丢失数据。
- Unreliable Receiver：当worker或者driver节点故障，这种receiver会丢失数据

选择哪种类型的receiver依赖于这些语义。如果一个worker节点出现故障，Reliable Receiver不会丢失数据，Unreliable Receiver会丢失接收了但是没有复制的数据。如果driver节点 出现故障，除了以上情况下的数据丢失，所有过去接收并复制到内存中的数据都会丢失，这会影响有状态transformation的结果。

为了避免丢失过去接收的数据，Spark 1.2引入了一个实验性的特征 `write ahead logs` ，它保存接收的数据到容错存储系统中。有了 `write ahead logs` 和Reliable Receiver，我们可以 做到零数据丢失以及exactly-once语义。

下面的表格总结了错误语义：

Deployment Scenario	Worker Failure	Driver Failure
Spark 1.1 或者更早, 没有write ahead log的Spark 1.2	在Unreliable Receiver情况下缓冲数据丢失；在Reliable Receiver和文件的情况下，零数据丢失	在Unreliable Receiver情况下缓冲数据丢失；在所有receiver情况下，过去的数据丢失；在文件的情况下，零数据丢失
带有write ahead log的Spark 1.2	在Reliable Receiver和文件的情况下，零数据丢失	在Reliable Receiver和文件的情况下，零数据丢失

输出操作的语义

根据其确定操作的谱系，所有数据都被建模成了RDD，所有的重新计算都会产生同样的结果。所有的DStream transformation都有exactly-once语义。那就是说，即使某个worker节点出现 故障，最终的转换结果都是一样。然而，输出操作（如 `foreachRDD` ）具有 `at-least once` 语义，那就是说，在有worker事件故障的情况下，变换后的数据可能被写入到一个外部实体不止一次。 利用 `saveAs***Files` 将数据保存到HDFS中的情况下，以上写多次是能够被接受的（因为文件会被相同的数据覆盖）。

Spark SQL

Spark SQL允许Spark执行用SQL, HiveQL或者Scala表示的关系查询。这个模块的核心是一个新类型的RDD-SchemaRDD。 SchemaRDDs由行对象组成，行对象拥有一个模式（scheme） 来描述行中每一列的数据类型。SchemaRDD与关系型数据库中的表很相似。可以通过存在的RDD、一个Parquet文件、一个JSON数据库或者对存储在Apache Hive中的数据执行HiveSQL查询中创建。

本章的所有例子都利用了Spark分布式系统中的样本数据，可以在 `spark-shell` 中运行它们。

开始

Spark中所有相关功能的入口点是SQLContext类或者它的子类， 创建一个SQLContext的所有需要仅仅是一个SparkContext。

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.
import sqlContext.createSchemaRDD
```

除了一个基本的SQLContext，你也能够创建一个HiveContext，它支持基本SQLContext所支持功能的一个超集。它的额外的功能包括用更完整的HiveQL分析器写查询去访问HiveUDFs的能力、 从Hive表读取数据的能力。用HiveContext你不需要一个已经存在的Hive开启，SQLContext可用的数据源对HiveContext也可用。HiveContext分开打包是为了避免在Spark构建时包含了所有 的Hive依赖。如果对你的应用程序来说，这些依赖不存在问题，Spark 1.2推荐使用HiveContext。以后的稳定版本将专注于为SQLContext提供与HiveContext等价的功能。

用来解析查询语句的特定SQL变种语言可以通过 `spark.sql.dialect` 选项来选择。这个参数可以通过两种方式改变，一种方式是通过 `setConf` 方法设定，另一种方式是在SQL命令中通过 `SET key=value` 来设定。对于SQLContext，唯一可用的方言是“sql”，它是Spark SQL提供的一个简单的SQL解析器。在HiveContext中，虽然也支持"sql"，但默认的方言是“hiveql”。这是因为HiveQL解析器更 完整。在很多用例中推荐使用“hiveql”。

数据源

Spark SQL支持通过SchemaRDD接口操作各种数据源。一个SchemaRDD能够作为一个一般的RDD被操作，也可以被注册为一个临时的表。注册一个SchemaRDD为一个表就 可以允许你在其数据上运行SQL查询。这节描述了加载数据为SchemaRDD的多种方法。

- RDDs
- parquet文件
- JSON数据集
- Hive表

RDDs

Spark支持两种方法将存在的RDDs转换为SchemaRDDs。第一种方法使用反射来推断包含特定对象类型的RDD的模式(schema)。在你写spark程序的同时，当你已经知道了模式，这种基于反射的 方法可以使代码更简洁并且程序工作得更好。

创建SchemaRDDs的第二种方法是通过一个编程接口来实现，这个接口允许你构造一个模式，然后在存在的RDDs上使用它。虽然这种方法更冗长，但是它允许你在运行期之前不知道列以及列 的类型的情况下构造SchemaRDDs。

利用反射推断模式

Spark SQL的Scala接口支持将包含样本类的RDDs自动转换为SchemaRDD。这个样本类定义了表的模式。

给样本类的参数名字通过反射来读取，然后作为列的名字。样本类可以嵌套或者包含复杂的类型如序列或者数组。这个RDD可以隐式转化为一个SchemaRDD，然后注册为一个表。表可以在后续的 sql语句中使用。

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.
import sqlContext.createSchemaRDD

// Define the schema using a case class.
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this
// you can use custom classes that implement the Product interface.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(",")).map(
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are SchemaRDDs and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

编程指定模式

当样本类不能提前确定（例如，记录的结构是经过编码的字符串，或者一个文本集合将会被解析，不同的字段投影给不同的用户），一个SchemaRDD可以通过三步来创建。

- 从原来的RDD创建一个行的RDD
- 创建由一个 `StructType` 表示的模式与第一步创建的RDD的行结构相匹配
- 在行RDD上通过 `applySchema` 方法应用模式

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Create an RDD
val people = sc.textFile("examples/src/main/resources/people.txt")

// The schema is encoded in a string
val schemaString = "name age"

// Import Spark SQL data types and Row.
import org.apache.spark.sql._

// Generate the schema based on the string of schema
val schema =
  StructType(
    schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))

// Convert records of the RDD (people) to Rows.
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

// Apply the schema to the RDD.
val peopleSchemaRDD = sqlContext.applySchema(rowRDD, schema)

// Register the SchemaRDD as a table.
peopleSchemaRDD.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val results = sqlContext.sql("SELECT name FROM people")

// The results of SQL queries are SchemaRDDs and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

Parquet文件

Parquet是一种柱状(columnar)格式，可以被许多其它的数据处理系统支持。Spark SQL提供支持读和写Parquet文件的功能，这些文件可以自动地保留原始数据的模式。

加载数据

```
// sqlContext from the previous example is used in this example.
// createSchemaRDD is used to implicitly convert an RDD to a SchemaRDD.
import sqlContext.createSchemaRDD

val people: RDD[Person] = ... // An RDD of case class objects, from the previous example.

// The RDD is implicitly converted to a SchemaRDD by createSchemaRDD, allowing it to be s
people.saveAsParquetFile("people.parquet")

// Read in the parquet file created above.  Parquet files are self-describing so the sche
// The result of loading a Parquet file is also a SchemaRDD.
val parquetFile = sqlContext.parquetFile("people.parquet")

//Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerTempTable("parquetFile")
val teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 1
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

配置

可以在SQLContext上使用setConf方法配置Parquet或者在用SQL时运行 `SET key=value` 命令来配置Parquet。

Property Name	Default	Meaning
spark.sql.parquet.binaryAsString	false	一些其它的Parquet-producing系统，特别是Impala和其它版本的Spark SQL，当写出Parquet模式的时候，二进制数据和字符串之间无法区分。这个标记告诉Spark SQL将二进制数据解释为字符串来提供这些系统的兼容性。
spark.sql.parquet.cacheMetadata	true	打开parquet元数据的缓存，可以提高静态数据的查询速度
spark.sql.parquet.compression.codec	gzip	设置写parquet文件时的压缩算法，可以接受的值包括：uncompressed, snappy, gzip, lzo
spark.sql.parquet.filterPushdown	false	打开Parquet过滤器的pushdown优化。因为已知的Paruet错误，这个特征默认是关闭的。如果你的表不包含任何空的字符串或者二进制列，打开这个特征仍是安全的
spark.sql.hive.convertMetastoreParquet	true	当设置为false时，Spark SQL将使用Hive SerDe代替内置的支持

JSON数据集

Spark SQL能够自动推断JSON数据集的模式，加载它为一个SchemaRDD。这种转换可以通过下面两种方法来实现

- jsonFile ：从一个包含JSON文件的目录中加载。文件中的每一行是一个JSON对象
- jsonRDD ：从存在的RDD加载数据，这些RDD的每个元素是一个包含JSON对象的字符串

注意，作为jsonFile的文件不是一个典型的JSON文件，每行必须是独立的并且包含一个有效的JSON对象。结果是，一个多行的JSON文件经常会失败

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files.
val path = "examples/src/main/resources/people.json"
// Create a SchemaRDD from the file(s) pointed to by path
val people = sqlContext.jsonFile(path)

// The inferred schema can be visualized using the printSchema() method.
people.printSchema()
// root
//  |-- age: integer (nullable = true)
//  |-- name: string (nullable = true)

// Register this SchemaRDD as a table.
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// Alternatively, a SchemaRDD can be created for a JSON dataset represented by
// an RDD[String] storing one JSON object per string.
val anotherPeopleRDD = sc.parallelize(
  """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

Hive表

Spark SQL也支持从Apache Hive中读出和写入数据。然而，Hive有大量的依赖，所以它不包含在Spark集合中。可以通过 `-Phive` 和 `-Phive-thriftserver` 参数构建Spark，使其 支持Hive。注意这个重新构建的jar包必须存在于所有的worker节点中，因为它们需要通过Hive的序列化和反序列化库访问存储在Hive中的数据。

当和Hive一起工作是，开发者需要提供HiveContext。HiveContext从SQLContext继承而来，它增加了在MetaStore中发现表以及利用HiveSql写查询的功能。没有Hive部署的用户也 可以创建HiveContext。当没有通过 `hive-site.xml` 配置，上下文将会在当前目录自动地创建 `metastore_db` 和 `warehouse` 。

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)

sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE s

// Queries are expressed in HiveQL
sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

性能调优

对于某些工作负载，可以在通过在内存中缓存数据或者打开一些实验选项来提高性能。

在内存中缓存数据

Spark SQL可以通过调用 `sqlContext.cacheTable("tableName")` 方法来缓存使用柱状格式的表。然后，Spark将会仅仅浏览需要的列并且自动地压缩数据以减少内存的使用以及垃圾回收的 压力。你可以通过调用 `sqlContext.uncacheTable("tableName")` 方法在内存中删除表。

注意，如果你调用 `schemaRDD.cache()` 而不是 `sqlContext.cacheTable(...)` ,表将不会用柱状格式来缓存。在这种情况下， `sqlContext.cacheTable(...)` 是强烈推荐的用法。

可以在SQLContext上使用setConf方法或者在用SQL时运行 `SET key=value` 命令来配置内存缓存。

Property Name	Default	Meaning
spark.sql.inMemoryColumnarStorage.compressed	true	当设置为true时，Spark SQL将为基于数据统计信息的每列自动选择一个压缩算法。
spark.sql.inMemoryColumnarStorage.batchSize	10000	柱状缓存的批数据大小。更大的批数据可以提高内存的利用率以及压缩效率，但有OOMs的风险

其它的配置选项

以下的选项也可以用来调整查询执行的性能。有可能这些选项会在以后的版本中弃用，这是因为更多的优化会自动执行。

Property Name	Default	Meaning
spark.sql.autoBroadcastJoinThreshold	10485760(10m)	配置一个表的最大大小(byte)。当执行join操作时，这个表将会广播到所有的worker节点。可以将值设置为-1来禁用广播。注意，目前的统计数据只支持Hive Metastore表，命令 `ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan` 已经在这个表中运行。
spark.sql.codegen	false	当为true时，特定查询中的表达式求值的代码将会在运行时动态生成。对于一些拥有复杂表达式的查询，此选项可导致显著速度提升。然而，对于简单的查询，这个选项会减慢查询的执行
spark.sql.shuffle.partitions	200	配置join或者聚合操作shuffle数据时分区的数量

其它SQL接口

Spark SQL也支持直接运行SQL查询的接口，不用写任何代码。

运行Thrift JDBC/ODBC服务器

这里实现的Thrift JDBC/ODBC服务器与Hive 0.12中的HiveServer2相一致。你可以用在Spark或者Hive 0.12附带的beeline脚本测试JDBC服务器。

在Spark目录中，运行下面的命令启动JDBC/ODBC服务器。

```
./sbin/start-thriftserver.sh
```

这个脚本接受任何的 `bin/spark-submit` 命令行参数，加上一个 `--hiveconf` 参数用来指明Hive属性。你可以运行 `./sbin/start-thriftserver.sh --help` 来获得所有可用选项的完整 列表。默认情况下，服务器监听 `localhost:10000` 。你可以用环境变量覆盖这些变量。

```
export HIVE_SERVER2_THRIFT_PORT=
export HIVE_SERVER2_THRIFT_BIND_HOST=
./sbin/start-thriftserver.sh \
  --master  \
  ...
```

或者通过系统变量覆盖。

```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port= \
  --hiveconf hive.server2.thrift.bind.host= \
  --master
  ...
```

现在你可以用beeline测试Thrift JDBC/ODBC服务器。

```
./bin/beeline
```

连接到Thrift JDBC/ODBC服务器的方式如下：

```
beeline> !connect jdbc:hive2://localhost:10000
```

Beeline将会询问你用户名和密码。在非安全的模式，简单地输入你机器的用户名和空密码就行了。对于安全模式，你可以按照Beeline文档的说明来执行。

运行Spark SQL CLI

Spark SQL CLI是一个便利的工具，它可以在本地运行Hive元存储服务、执行命令行输入的查询。注意，Spark SQL CLI不能与Thrift JDBC服务器通信。

在Spark目录运行下面的命令可以启动Spark SQL CLI。

```
./bin/spark-sql
```

编写语言集成(Language-Integrated)的相关查询

语言集成的相关查询是实验性的，现在暂时只支持scala。

Spark SQL也支持用领域特定语言编写查询。

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// Importing the SQL context gives access to all the public SQL functions and implicit co
import sqlContext._
val people: RDD[Person] = ... // An RDD of case class objects, from the first example.

// The following is the same as 'SELECT name FROM people WHERE age >= 10 AND age <= 19'
val teenagers = people.where('age >= 10).where('age <= 19).select('name)
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

DSL使用Scala的符号来表示在潜在表(underlying table)中的列，这些列以前缀(')标示。将这些符号隐式转换成由SQL执行引擎计算的表达式。你可以在ScalaDoc 中了解详情。

Spark SQL数据类型

- 数字类型
    - ByteType：代表一个字节的整数。范围是-128到127
    - ShortType：代表两个字节的整数。范围是-32768到32767
    - IntegerType：代表4个字节的整数。范围是-2147483648到2147483647
    - LongType：代表8个字节的整数。范围是-9223372036854775808到9223372036854775807
    - FloatType：代表4字节的单精度浮点数
    - DoubleType：代表8字节的双精度浮点数
    - DecimalType：代表任意精度的10进制数据。通过内部的java.math.BigDecimal支持。BigDecimal由一个任意精度的整型非标度值和一个32位整数组成
    - StringType：代表一个字符串值
    - BinaryType：代表一个byte序列值
    - BooleanType：代表boolean值
    - Datetime类型
        - TimestampType：代表包含字段年，月，日，时，分，秒的值
        - DateType：代表包含字段年，月，日的值
    - 复杂类型
        - ArrayType(elementType, containsNull)：代表由elementType类型元素组成的序列值。`containsNull` 用来指明 `ArrayType` 中的值是否有null值
        - MapType(keyType, valueType, valueContainsNull)：表示包括一组键 - 值对的值。通过keyType表示key数据的类型，通过valueType表示value数据的类型。`valueContainsNull` 用来指明 `MapType` 中的值是否有null值
        - StructType(fields):表示一个拥有 `StructFields (fields)` 序列结构的值
            - StructField(name, dataType, nullable):代表 `StructType` 中的一个字段，字段的名字通过 `name` 指定，`dataType` 指定field的数据类型，`nullable` 表示字段的值是否有null值。

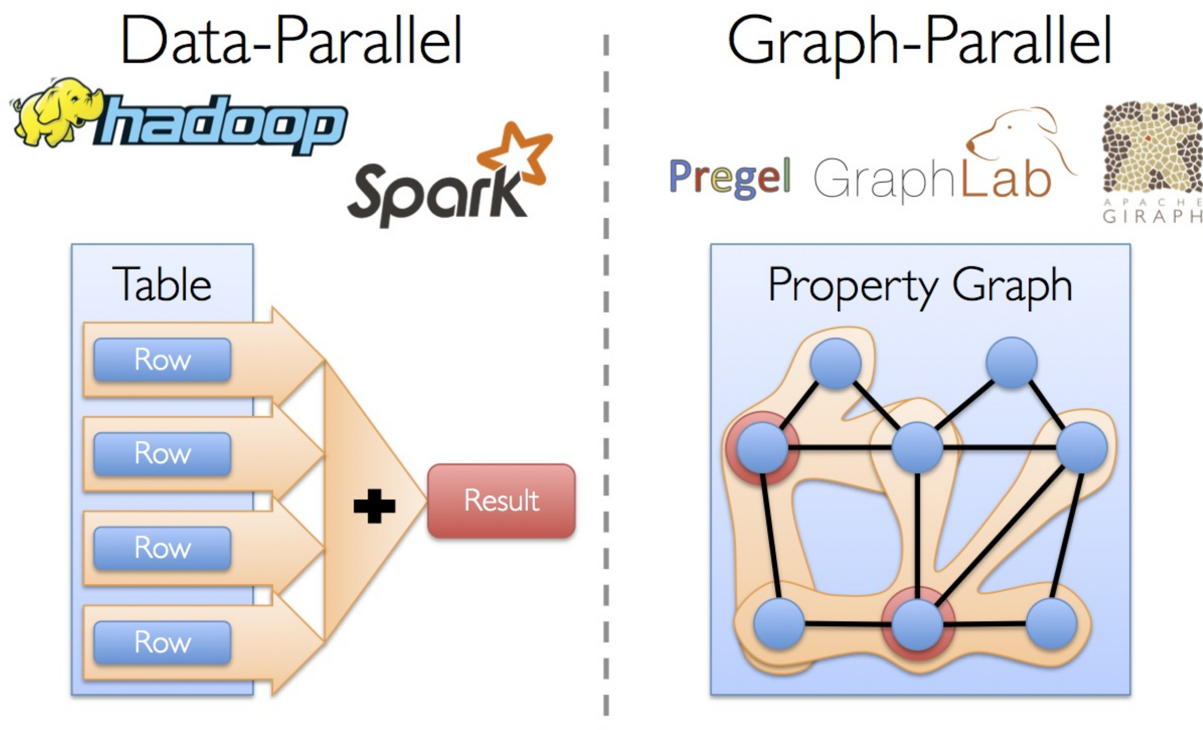
Spark的所有数据类型都定义在包 `org.apache.spark.sql` 中，你可以通过 `import org.apache.spark.sql._` 访问它们。

数据类型	Scala中的值类型	访问或者创建数据类型的API
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	scala.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull]) 注意containsNull默认为true
MapType	scala.collection.Map	MapType(keyType, valueType, [valueContainsNull]) 注意valueContainsNull默认为true
StructType	org.apache.spark.sql.Row	StructType(fields)， 注意fields是一个StructField序列，相同名字的两个StructField不被允许
StructField	The value type in Scala of the data type of this field (For example, Int for a StructField with the data type IntegerType)	StructField(name, dataType, nullable)

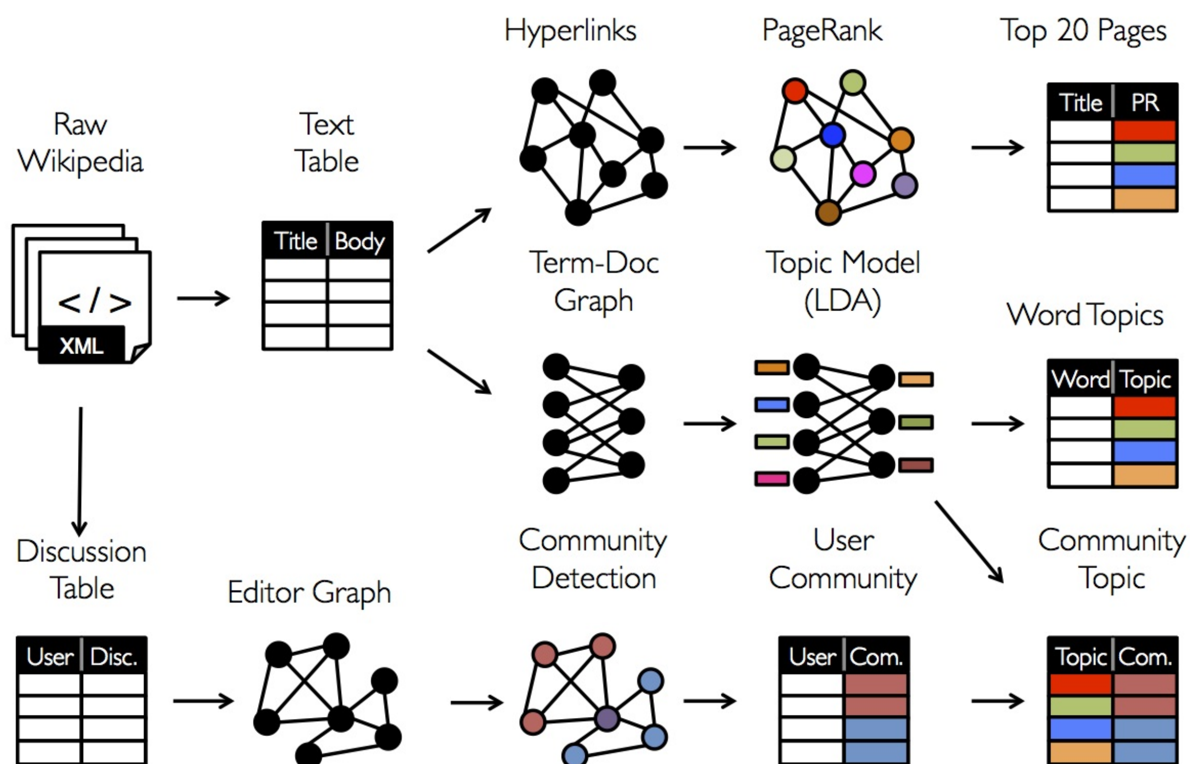
GraphX编程指南

GraphX是一个新的（alpha）Spark API，它用于图和并行图(graph-parallel)的计算。GraphX通过引入Resilient Distributed Property Graph：带有 顶点和边属性的有向多重图，来扩展Spark RDD。为了支持图计算，GraphX公开一组基本的功能操作以及Pregel API的一个优化。另外，GraphX包含了一个日益增长的图算法和图builders的 集合，用以简化图分析任务。

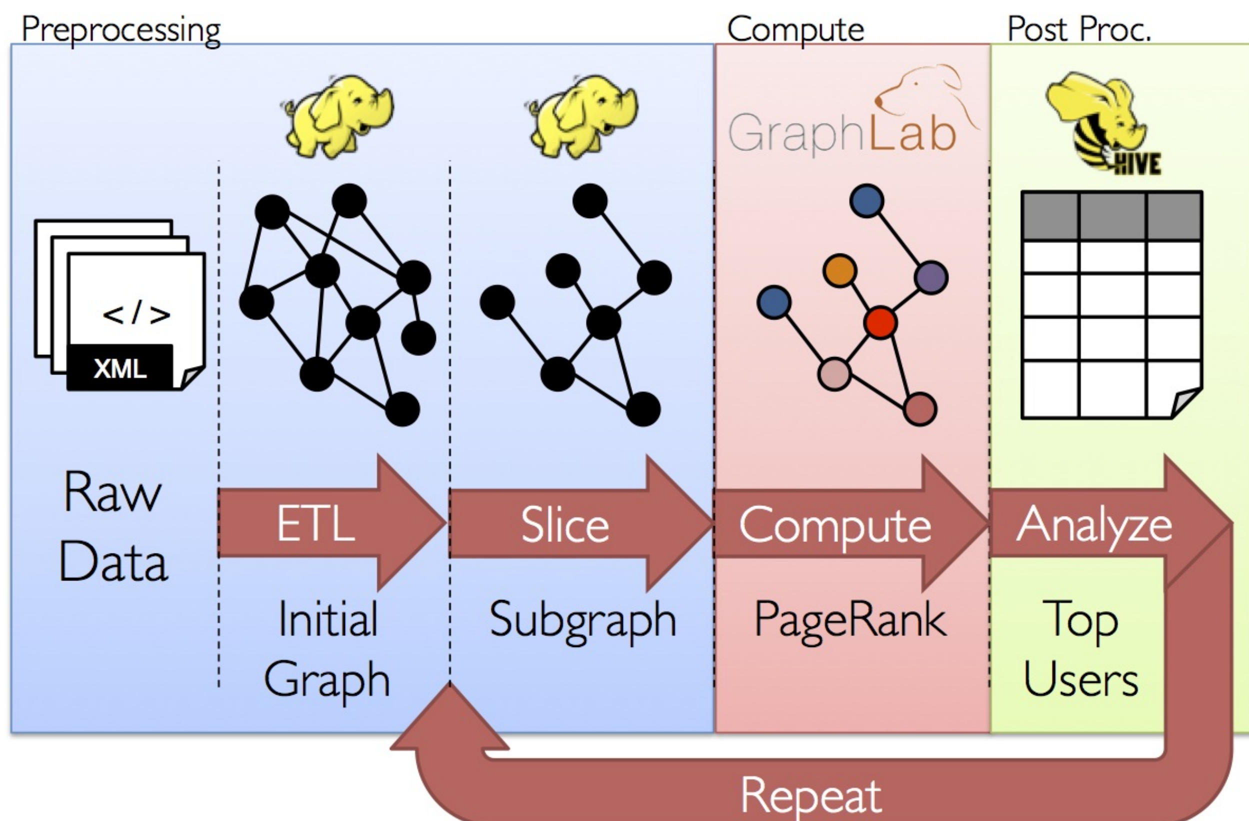
从社交网络到语言建模，不断增长的规模和图形数据的重要性已经推动了许多新的`graph-parallel`系统（如Giraph和GraphLab）的发展。 通过限制可表达的计算类型和引入新的技术来划分和分配图，这些系统可以高效地执行复杂的图形算法，比一般的`data-parallel`系统快很多。



然而，通过这种限制可以提高性能，但是很难表示典型的图分析途径（构造图、修改它的结构或者表示跨多个图的计算）中很多重要的stages。另外，我们如何看待数据取决于我们的目标，并且同一原始数据可能有许多不同表和图的视图。



结论是，图和表之间经常需要能够相互移动。然而，现有的图分析管道必须组成 `graph-parallel` 和 `data- parallel` 系统`，从而实现大数据的迁移和复制并生成一个复杂的编程模型。



GraphX项目的目的就是将 `graph-parallel` 和 `data-parallel` 统一到一个系统中，这个系统拥有一个唯一的组合API。GraphX允许用户将数据当做一个图和一个集合（RDD），而不需要而不需要数据移动或者复杂。通过将最新的进展整合进 `graph-parallel` 系统，GraphX能够优化图操作的执行。

开始

开始的第一步是引入Spark和GraphX到你的项目中，如下面所示

```
mport org.apache.spark._
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD
```

如果你没有用到Spark shell，你还将需要SparkContext。

属性图

属性图是一个有向多重图，它带有连接到每个顶点和边的用户定义的对象。 有向多重图中多个并行(parallel)的边共享相同的源和目的地顶点。支持并行边的能力简化了建模场景，这个场景中，相同的顶点存在多种关系(例如co-worker和friend)。每个顶点由一个 唯一的64位长的标识符（VertexID）作为key。GraphX并没有对顶点标识强加任何排序。同样，顶点拥有相应的源和目的顶点标识符。

属性图通过vertex(VD)和edge(ED)类型参数化，这些类型是分别与每个顶点和边相关联的对象的类型。

在某些情况下，在相同的图形中，可能希望顶点拥有不同的属性类型。这可以通过继承完成。例如，将用户和产品建模成一个二分图，我们可以用如下方式

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

和RDD一样，属性图是不可变的、分布式的、容错的。图的值或者结构的改变需要按期望的生成一个新的图来实现。注意，原始图的大部分都可以在新图中重用，用来减少这种固有的功能数据结构的成本。 执行者使用一系列顶点分区试探法来对图进行分区。如RDD一样，图中的每个分区可以在发生故障的情况下被重新创建在不同的机器上。

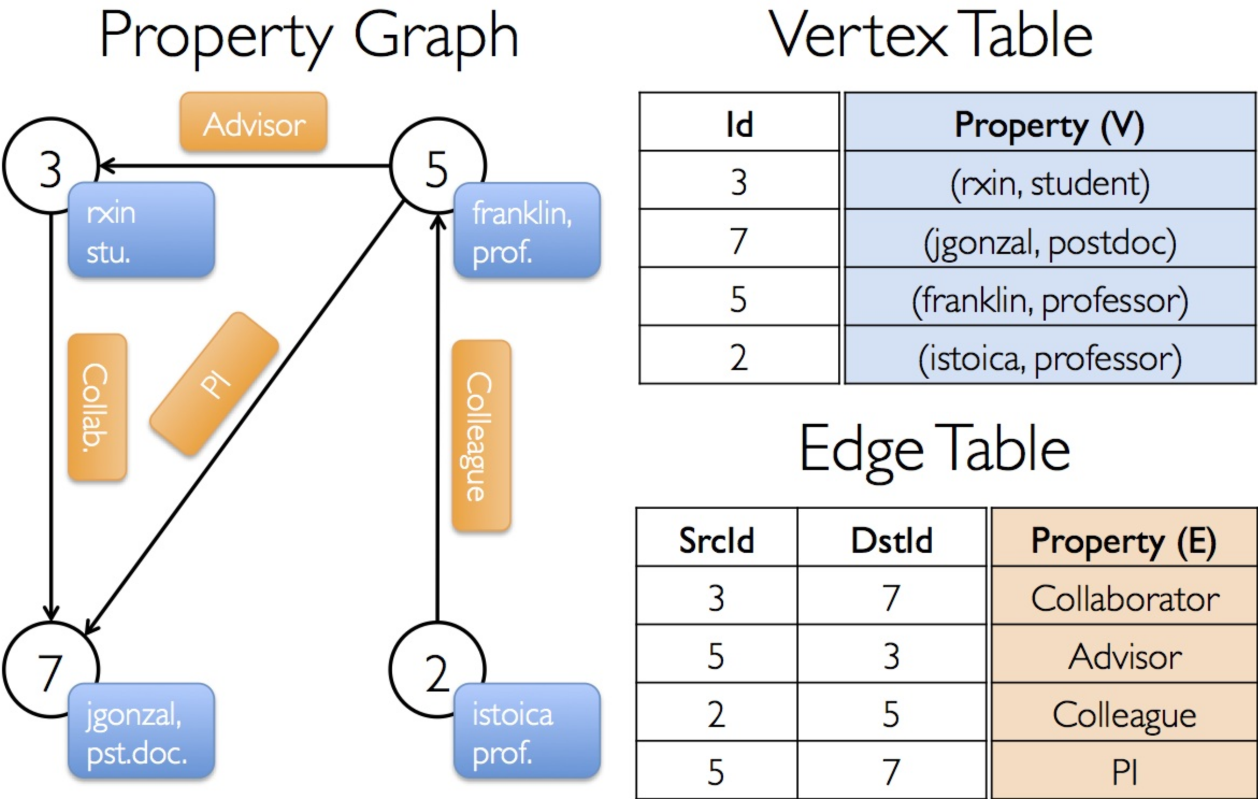
逻辑上的属性图对应于一对类型化的集合(RDD),这个集合编码了每一个顶点和边的属性。因此，图类包含访问图中顶点和边的成员。

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

`VertexRDD[VD]` 和 `EdgeRDD[ED]` 类分别继承和优化自 `RDD[(VertexID, VD)]` 和 `RDD[Edge[ED]]` 。 `VertexRDD[VD]` 和 `EdgeRDD[ED]` 都支持额外的功能来建立在图计算和利用内部优化。

属性图的例子

在GraphX项目中，假设我们想构造一个包括不同合作者的属性图。顶点属性可能包含用户名和职业。我们可以用描述合作者之间关系的字符串标注边缘。



所得的图形将具有类型签名

```
val userGraph: Graph[(String, String), String]
```

有很多方式从一个原始文件、RDD构造一个属性图。最一般的方法是利用Graph object。 下面的代码从RDD集合生成属性图。

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

在上面的例子中，我们用到了Edge样本类。边有一个 `srcId` 和 `dstId` 分别对应 于源和目标顶点的标示符。另外， `Edge` 类有一个 `attr` 成员用来存储边属性。

我们可以分别用 `graph.vertices` 和 `graph.edges` 成员将一个图解构为相应的顶点和边。

```
val graph: Graph[(String, String), String] // Constructed from above
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

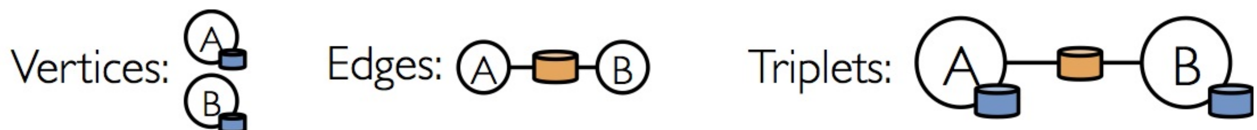
```
注意，graph.vertices返回一个VertexRDD[(String, String)]，它继承于 RDD[(VertexID, (String, Str
graph.edges返回一个包含Edge[String]对象的EdgeRDD。我们也可以用到case类的类型构造器，如下例所示。

graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

除了属性图的顶点和边视图，GraphX也包含了一个三元组视图，三元视图逻辑上将顶点和边的属性保存为一个 `RDD[EdgeTriplet[VD, ED]]` ，它包含EdgeTriplet类的实例。 可以通过下面的Sql表达式表示这个连接。

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

或者通过下面的图来表示。



`EdgeTriplet` 类继承于 `Edge` 类，并且加入了 `srcAttr` 和 `dstAttr` 成员，这两个成员分别包含源和目的的属性。我们可以用一个三元组视图渲染字符串集合用来描述用户之间的关系。

```
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

图操作符

正如RDDs有基本的操作map, filter和reduceByKey一样，属性图也有基本的集合操作，这些操作采用用户自定义的函数并产生包含转换特征和结构的新图。定义在Graph中的 核心操作是经过优化的实现。表示为核心操作的组合的便捷操作定义在GraphOps中。然而， 因为有Scala的隐式转换，定义在 `GraphOps` 中的操作可以作为 `Graph` 的成员自动使用。例如，我们可以通过下面的方式计算每个顶点(定义在GraphOps中)的入度。

```
val graph: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees: VertexRDD[Int] = graph.inDegrees
```

区分核心图操作和 `GraphOps` 的原因是为了在将来支持不同的图表示。每个图表示都必须提供核心操作的实现并重用很多定义在 `GraphOps` 中的有用操作。

操作一览

一下是定义在 `Graph` 和 `GraphOps` 中（为了简单起见，表现为图的成员）的功能的快速浏览。注意，某些函数签名已经简化（如默认参数和类型的限制已删除），一些更高级的功能已经被 删除，所以请参阅API文档了解官方的操作列表。

```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // Information about the Graph ===========================================================
  val numEdges: Long
  val numVertices: Long
  val inDegrees: VertexRDD[Int]
  val outDegrees: VertexRDD[Int]
  val degrees: VertexRDD[Int]
  // Views of the graph as collections =====================================================
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  val triplets: RDD[EdgeTriplet[VD, ED]]
  // Functions for caching graphs ==========================================================
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
  // Change the partitioning heuristic  ====================================================
  def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
  // Transform vertex and edge attributes ==================================================
  def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, E
```

```
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2]
    : Graph[VD, ED2]
  // Modify the graph structure ====================================================
  def reverse: Graph[VD, ED]
  def subgraph(
      epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
      vpred: (VertexID, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
  // Join RDDs with the graph ======================================================
  def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Graph
  def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])
      (mapFunc: (VertexID, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
  // Aggregate information about adjacent triplets =================================
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexID]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexID, VD)]]
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[A]
  // Iterative graph-parallel computation ==========================================
  def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
      vprog: (VertexID, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID,A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED]
  // Basic graph algorithms ========================================================
  def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
  def connectedComponents(): Graph[VertexID, ED]
  def triangleCount(): Graph[Int, ED]
  def stronglyConnectedComponents(numIter: Int): Graph[VertexID, ED]
}
```

属性操作

如RDD的 `map` 操作一样，属性图包含下面的操作：

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

每个操作都产生一个新的图，这个新的图包含通过用户自定义的map操作修改后的顶点或边的属性。

注意，每种情况下图结构都不受影响。这些操作的一个重要特征是它允许所得图形重用原有图形的结构索引(indices)。下面的两行代码在逻辑上是等价的，但是第一个不保存结构索引，所以 不会从GraphX系统优化中受益。

```
val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr)) }
val newGraph = Graph(newVertices, graph.edges)
```

另一种方法是用mapVertices⇒VD2)(ClassTag[VD2]):Graph[VD2,ED])保存索引。

```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

这些操作经常用来初始化的图形，用作特定计算或者用来处理项目不需要的属性。例如，给定一个图，这个图的顶点特征包含出度，我们为PageRank初始化它。

```
// Given a graph where the vertex property is the out degree
val inputGraph: Graph[Int, String] =
  graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) => degOpt.getOrElse(0))
// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
val outputGraph: Graph[Double, Double] =
  inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _) => 1.0)
```

结构性操作

当前的GraphX仅仅支持一组简单的常用结构性操作。下面是基本的结构性操作列表。

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

reverse操作返回一个新的图，这个图的边的方向都是反转的。例如，这个操作可以用来计算反转的PageRank。因为反转操作没有修改顶点或者边的属性或者改变边的数量，所以我们可以 在不移动或者复制数据的情况下有效地实现它。

subgraph⇒Boolean,(VertexId,VD)⇒Boolean):Graph[VD,ED])操作 利用顶点和边的谓词（predicates），返回的图仅仅包含满足顶点谓词的顶点、满足边谓词的边以及满足顶点谓词的连接顶点（connect vertices）。 `subgraph` 操作可以用于很多场景，如获取 感兴趣的顶点和边组成的图或者获取清除断开链接后的图。下面的例子删除了断开的链接。

```
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
                       (4L, ("peter", "student"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"),
                       Edge(4L, 0L, "student"),   Edge(5L, 0L, "colleague")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
// Notice that there is a user 0 (for which we have no information) connected to users
// 4 (peter) and 5 (franklin).
graph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr.
  ).collect.foreach(println(_))
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// The valid subgraph will disconnect users 4 and 5 by removing user 0
validGraph.vertices.collect.foreach(println(_))
validGraph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr.
  ).collect.foreach(println(_))
```

注意，上面的例子中，仅仅提供了顶点谓词。如果没有提供顶点或者边的谓词， `subgraph` 操作默认为true。

mask(ClassTag[VD2],ClassTag[ED2]):Graph[VD,ED])操作 构造一个子图，这个子图包含输入图中包含的顶点和边。这个操作可以和 `subgraph` 操作相结合，基于另外一个相关图的特征去约束一个图。例如，我们可能利用缺失顶点的图运行连通体（？连通组件connected components），然后返回有效的子图。

```
/ Run Connected Components
val ccGraph = graph.connectedComponents() // No longer contains missing field
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

groupEdges⇒ED):Graph[VD,ED])操作合并多重图 中的并行边(如顶点对之间重复的边)。在大量的应用程序中，并行的边可以合并（它们的权重合并）为一条边从而降低图的大小。

连接操作

在许多情况下，有必要将外部数据加入到图中。例如，我们可能有额外的用户属性需要合并到已有的图中或者我们可能想从一个图中取出顶点特征加入到另外一个图中。这些任务可以用join操作完成。 下面列出的是主要的join操作。

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U])
    : Graph[VD2, ED]
}
```

joinVertices])((VertexId,VD,U)⇒VD)(ClassTag[U]):Graph[VD,ED]) 操作将输入RDD和顶点相结合，返回一个新的带有顶点特征的图。这些特征是通过在连接顶点的结果上使用用户定义的 `map` 函数获得的。在RDD中没有匹配值的顶点保留其原始值。

注意，对于给定的顶点，如果RDD中有超过1个的匹配值，则仅仅使用其中的一个。建议用下面的方法保证输入RDD的唯一性。下面的方法也会预索引返回的值用以加快后续的join操作。

```
val nonUniqueCosts: RDD[(VertexID, Double)]
val uniqueCosts: VertexRDD[Double] =
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```

除了将用户自定义的map函数用到所有顶点和改变顶点属性类型以外，更一般的 outerJoinVertices])((VertexId,VD,Option[U])⇒VD2)(ClassTag[U],ClassTag[VD2]):Graph[VD2,ED])与 `joinVertices` 类似。 因为并不是所有顶点在RDD中拥有匹配的值，map函数需要一个option类型。

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt) =>
  outDegOpt match {
    case Some(outDeg) => outDeg
    case None => 0 // No outDegree means zero outDegree
  }
}
```

你可能已经注意到了，在上面的例子中用到了curry函数的多参数列表。虽然我们可以将f(a)(b)写成f(a,b)，但是f(a,b)意味着b的类型推断将不会依赖于a。因此，用户需要为定义 的函数提供类型标注。

```
val joinedGraph = graph.joinVertices(uniqueCosts,
  (id: VertexID, oldCost: Double, extraCost: Double) => oldCost + extraCost)
```

相邻聚合（Neighborhood Aggregation）

图分析任务的一个关键步骤是汇总每个顶点附近的信息。例如我们可能想知道每个用户的追随者的数量或者每个用户的追随者的平均年龄。许多迭代图算法（如PageRank，最短路径和连通体） 多次聚合相邻顶点的属性。

为了提高性能，主要的聚合操作从 `graph.mapReduceTriplets` 改为了新的 `graph.AggregateMessages` 。虽然API的改变相对较小，但是我们仍然提供了过渡的指南。

聚合消息(aggregateMessages)

GraphX中的核心聚合操作是aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)(ClassTag[A]):VertexRDD[A])。 这个操作将用户定义的 `sendMsg` 函数应用到图的每个边三元组(edge triplet)，然后应用 `mergeMsg` 函数在其目的顶点聚合这些消息。

```
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
}
```

用户自定义的 `sendMsg` 函数是一个EdgeContext类型。它暴露源和目的属性以及边缘属性 以及发送消息给源和目的属性的函数(sendToSrc:Unit)和sendToDst:Unit))。 可将 `sendMsg` 函数看做map-reduce过程中的map函数。用户自定义的 `mergeMsg` 函数指定两个消息到相同的顶点并保存为一个消息。可以将 `mergeMsg` 函数看做map-reduce过程中的reduce函数。aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)(ClassTag[A]):VertexRDD[A]) 操作返回一个包含聚合消息(目的地为每个顶点)的 `VertexRDD[Msg]` 。没有接收到消息的顶点不包含在返回的 `VertexRDD` 中。

另外，aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)(ClassTag[A]):VertexRDD[A]) 有一个可选的 `tripletFields` 参数，它指出在 `EdgeContext` 中，哪些数据被访问（如源顶点特征而不是目的顶点特征）。 `tripletsFields` 可能的选项定义在TripletFields中。 `tripletFields` 参数

用来通知GraphX仅仅只需要 `EdgeContext` 的一部分允许GraphX选择一个优化的连接策略。例如，如果我们想计算每个用户的追随者的平均年龄，我们仅仅只需要源字段。 所以我们用 `TripletFields.Src` 表示我们仅仅只需要源字段。

在下面的例子中，我们用 `aggregateMessages` 操作计算每个用户更年长的追随者的年龄。

```
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// Create a graph with "age" as the vertex property.  Here we use a random graph for simp
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.toDoub
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst(1, triplet.srcAttr)
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) => value match { case (count, totalAge) => totalA
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

当消息（以及消息的总数）是常量大小(列表和连接替换为浮点数和添加)时， `aggregateMessages` 操作的效果最好。

Map Reduce三元组过渡指南

在之前版本的GraphX中，利用[mapReduceTriplets]操作完成相邻聚合。

```
class Graph[VD, ED] {
  def mapReduceTriplets[Msg](
      map: EdgeTriplet[VD, ED] => Iterator[(VertexId, Msg)],
      reduce: (Msg, Msg) => Msg)
    : VertexRDD[Msg]
}
```

`mapReduceTriplets` 操作在每个三元组上应用用户定义的map函数，然后保存用用户定义的reduce函数聚合的消息。然而，我们发现用户返回的迭代器是昂贵的，它抑制了我们添加额外优化(例如本地顶点的重新编号)的能力。 aggregateMessages⇒Unit,(A,A)⇒A,TripletFields)

(ClassTag[A]):VertexRDD[A]) 暴露三元组字段和函数显示的发送消息到源和目的顶点。并且，我们删除了字节码检测转而需要用户指明三元组的哪些字段实际需要。

下面的代码用到了 `mapReduceTriplets`

```
val graph: Graph[Int, Float] = ...
def msgFun(triplet: Triplet[Int, Float]): Iterator[(Int, String)] = {
  Iterator((triplet.dstId, "Hi"))
}
def reduceFun(a: Int, b: Int): Int = a + b
val result = graph.mapReduceTriplets[String](msgFun, reduceFun)
```

下面的代码用到了 `aggregateMessages`

```
val graph: Graph[Int, Float] = ...
def msgFun(triplet: EdgeContext[Int, Float, String]) {
  triplet.sendToDst("Hi")
}
def reduceFun(a: Int, b: Int): Int = a + b
val result = graph.aggregateMessages[String](msgFun, reduceFun)
```

计算度信息

最一般的聚合任务就是计算顶点的度，即每个顶点相邻边的数量。在有向图中，经常需要知道顶点的入度、出度以及总共的度。GraphOps 类包含一个操作集合用来计算每个顶点的度。例如，下面的例子计算最大的入度、出度和总度。

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

Collecting Neighbors

在某些情况下，通过收集每个顶点相邻的顶点及它们的属性来表达计算可能更容易。这可以通过collectNeighborIds:VertexRDD[Array[VertexId]]) 和 collectNeighbors:VertexRDD[Array[(VertexId,VD)]])操作来简单的完成

```
class GraphOps[VD, ED] {
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[ Array[(VertexId, VD)] ]
}
```

这些操作是非常昂贵的，因为它们需要重复的信息和大量的通信。如果可能，尽量用 `aggregateMessages` 操作直接表达相同的计算。

缓存和不缓存

在Spark中，RDDs默认是不缓存的。为了避免重复计算，当需要多次利用它们时，我们必须显示地缓存它们。GraphX中的图也有相同的方式。当利用到图多次时，确保首先访问 `Graph.cache()` 方法。

在迭代计算中，为了获得最佳的性能，不缓存可能是必须的。默认情况下，缓存的RDDs和图会一直保留在内存中直到因为内存压力迫使它们以LRU的顺序删除。对于迭代计算，先前的迭代的中间结果将填充到缓存 中。虽然它们最终会被删除，但是保存在内存中的不需要的数据将会减慢垃圾回收。只有中间结果不需要，不缓存它们是更高效的。这涉及到在每次迭代中物化一个图或者RDD而不缓存所有其它的数据集。 在将来的迭代中仅用物化的数据集。然而，因为图是由多个RDD组成的，正确的不持久化它们是困难的。对于迭代计算，我们建议使用Pregel API，它可以正确的不持久化中间结果。

Pregel API

图本身是递归数据结构，顶点的属性依赖于它们邻居的属性，这些邻居的属性又依赖于自己邻居的属性。所以许多重要的图算法都是迭代的重新计算每个顶点的属性，直到满足某个确定的条件。 一系列的graph-parallel抽象已经被提出来用来表达这些迭代算法。GraphX公开了一个类似Pregel的操作，它是广泛使用的Pregel和GraphLab抽象的一个融合。

在GraphX中，更高级的Pregel操作是一个约束到图拓扑的批量同步（bulk-synchronous）并行消息抽象。Pregel操作者执行一系列的超级步骤（super steps），在这些步骤中，顶点从之前的超级步骤中接收进入(inbound)消息的总和，为顶点属性计算一个新的值，然后在以后的超级步骤中发送消息到邻居顶点。不像Pregel而更像GraphLab，消息作为一个边三元组的函数被并行 计算，消息计算既访问了源顶点特征也访问了目的顶点特征。在超级步中，没有收到消息的顶点被跳过。当没有消息遗留时，Pregel操作停止迭代并返回最终的图。

注意，与更标准的Pregel实现不同的是，GraphX中的顶点仅仅能发送信息给邻居顶点，并利用用户自定义的消息函数构造消息。这些限制允许在GraphX进行额外的优化。

一下是 Pregel操作((VertexId,VD,A)⇒VD,(EdgeTriplet[VD,ED])⇒Iterator[(VertexId,A)],(A,A)⇒A)(ClassTag[A]):Graph[VD,ED])的类型签名以及实现草图（注意，访问graph.cache已经被删除）

```
class GraphOps[VD, ED] {
  def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
    // Receive the initial message at each vertex
    var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()
    // compute the messages
    var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // Loop until no messages remain or maxIterations is achieved
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      // Receive the messages: -----------------------------------------------------------
      // Run the vertex program on all vertices that receive messages
      val newVerts = g.vertices.innerJoin(messages)(vprog).cache()
      // Merge the new vertex values back into the graph
      g = g.outerJoinVertices(newVerts) { (vid, old, newOpt) => newOpt.getOrElse(old) }.c
      // Send Messages: ------------------------------------------------------------------
      // Vertices that didn't receive a message above don't appear in newVerts and theref
      // get to send messages.  More precisely the map phase of mapReduceTriplets is only
      // on edges in the activeDir of vertices in newVerts
      messages = g.mapReduceTriplets(sendMsg, mergeMsg, Some((newVerts, activeDir))).cach
      activeMessages = messages.count()
      i += 1
    }
    g
  }
}
```

注意，pregel有两个参数列表（graph.pregel(list1)(list2)）。第一个参数列表包含配置参数初始消息、最大迭代数、发送消息的边的方向（默认是沿边方向出）。第二个参数列表包含用户自定义的函数用来接收消息（vprog）、计算消息（sendMsg）、合并消息（mergeMsg）。

我们可以用Pregel操作表达计算单源最短路径(single source shortest path)。

```
import org.apache.spark.graphx._
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// A graph with edge attributes containing distances
val graph: Graph[Int, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// Initialize the graph such that all vertices except the root have distance infinity.
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.Posit
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a,b) => math.min(a,b) // Merge Message
  )
println(sssp.vertices.collect.mkString("\n"))
```

图构造者

GraphX提供了几种方式从RDD或者磁盘上的顶点和边集合构造图。默认情况下，没有哪个图构造者为图的边重新分区，而是把边保留在默认的分区中（例如HDFS中它们的原始块）。Graph.groupEdges⇒ED):Graph[VD,ED]) 需要重新分区图，因为它假定相同的边将会被分配到同一个分区，所以你必须在调用groupEdges之前调用Graph.partitionBy:Graph[VD,ED])

```
object GraphLoader {
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}
```

GraphLoader.edgeListFile:Graph[Int,Int]) 提供了一个方式从磁盘上的边列表中加载一个图。它解析如下形式（源顶点ID，目标顶点ID）的连接表，跳过以 `#` 开头的注释行。

```
# This is a comment
2 1
4 1
1 2
```

它从指定的边创建一个图，自动地创建边提及的所有顶点。所有的顶点和边的属性默认都是1。 `canonicalOrientation` 参数允许重定向正方向(srcId < dstId)的边。这在connected components 算法中需要用到。 `minEdgePartitions` 参数指定生成的边分区的最少数量。边分区可能比指定的分区更多，例如，一个HDFS文件包含更多的块。

```
object Graph {
  def apply[VD, ED](
      vertices: RDD[(VertexId, VD)],
      edges: RDD[Edge[ED]],
      defaultVertexAttr: VD = null)
    : Graph[VD, ED]
  def fromEdges[VD, ED](
      edges: RDD[Edge[ED]],
      defaultValue: VD): Graph[VD, ED]
  def fromEdgeTuples[VD](
      rawEdges: RDD[(VertexId, VertexId)],
      defaultValue: VD,
      uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]
}
```

Graph.apply],RDD[Edge[ED]],VD)(ClassTag[VD],ClassTag[ED]):Graph[VD,ED]) 允许从顶点和边的RDD上创建一个图。重复的顶点可以任意的选择其中一个，在边RDD中而不是在顶点RDD中发现的顶点分配默认的属性。

Graph.fromEdges(ClassTag[VD],ClassTag[ED]):Graph[VD,ED]) 允许仅仅从一个边RDD上创建一个图，它自动地创建边提及的顶点，并分配这些顶点默认的值。

Graph.fromEdgeTuples],VD,Option[PartitionStrategy])(ClassTag[VD]):Graph[VD,Int]) 允许仅仅从一个边元组组成的RDD上创建一个图。分配给边的值为1。它自动地创建边提及的顶点，并分配这些顶点默认的值。它还支持删除边。为了删除边，需要传递一个PartitionStrategy 为值的 `Some` 作为 `uniqueEdges` 参数（如uniqueEdges = Some(PartitionStrategy.RandomVertexCut)）。分配相同的边到同一个分区从而使它们可以被删除，一个分区策略是必须的。

顶点和边RDDs

GraphX暴露保存在图中的顶点和边的RDD。然而，因为GraphX包含的顶点和边拥有优化的数据结构，这些数据结构提供了额外的功能。顶点和边分别返回 `VertexRDD` 和 `EdgeRDD` 。这一章 我们将学习它们的一些有用的功能。

VertexRDDs

`VertexRDD[A]` 继承自 `RDD[(VertexID, A)]` 并且添加了额外的限制，那就是每个 `VertexID` 只能出现一次。此外， `VertexRDD[A]` 代表了一组属性类型为A的顶点。在内部，这通过 保存顶点属性到一个可重复使用的hash-map数据结构来获得。所以，如果两个 `VertexRDDs` 从相同的基本 `VertexRDD` 获得（如通过filter或者mapValues），它们能够在固定的时间内连接 而不需要hash评价。为了利用这个索引数据结构， `VertexRDD` 暴露了一下附加的功能：

```
class VertexRDD[VD] extends RDD[(VertexID, VD)] {
  // Filter the vertex set but preserves the internal index
  def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
  // Transform the values without changing the ids (preserves the internal index)
  def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
  def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
  // Remove vertices from this set that appear in the other set
  def diff(other: VertexRDD[VD]): VertexRDD[VD]
  // Join operators that take advantage of the internal indexing to accelerate joins (sub
  def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD, Option[VD2]) => V
  def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2): VertexRD
  // Use the index on this RDD to accelerate a `reduceByKey` operation on the input RDD.
  def aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2, VD2) => VD2
}
```

举个例子， `filter` 操作如何返回一个VertexRDD。过滤器实际使用一个 `BitSet` 实现，因此它能够重用索引以及保留和其它 `VertexRDDs` 做连接时速度快的能力。同样的， `mapValues` 操作 不允许 `map` 函数改变 `VertexID` ，因此可以保证相同的 `HashMap` 数据结构能够重用。当连接两个从相同的 `hashmap` 获取的VertexRDDs和使用线性扫描而不是昂贵的点查找实现连接操作时， `leftJoin` 和 `innerJoin` 都能够使用。

从一个 `RDD[(VertexID, A)]` 高效地构建一个新的 `VertexRDD` ， `aggregateUsingIndex` 操作是有用的。概念上，如果我通过一组顶点构造了一个 `VertexRDD[B]` ，而 `VertexRDD[B]` 是 一些 `RDD[(VertexID, A)]` 中顶点的超集，那么我们就可以在聚合以及随后索引 `RDD[(VertexID, A)]` 中重用索引。例如：

```
val setA: VertexRDD[Int] = VertexRDD(sc.parallelize(0L until 100L).map(id => (id, 1)))
val rddB: RDD[(VertexId, Double)] = sc.parallelize(0L until 100L).flatMap(id => List((id,
// There should be 200 entries in rddB
rddB.count
val setB: VertexRDD[Double] = setA.aggregateUsingIndex(rddB, _ + _)
// There should be 100 entries in setB
setB.count
// Joining A and B should now be fast!
val setC: VertexRDD[Double] = setA.innerJoin(setB)((id, a, b) => a + b)
```

EdgeRDDs

`EdgeRDD[ED]` 继承自 `RDD[Edge[ED]]` ，使用定义在PartitionStrategy的 各种分区策略中的一个在块分区中组织边。在每个分区中，边属性和相邻结构被分别保存，当属性值改变时，它们可以最大化的重用。

`EdgeRDD` 暴露了三个额外的函数

```
// Transform the edge attributes while preserving the structure
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
// Revere the edges reusing both attributes and structure
def reverse: EdgeRDD[ED]
// Join two `EdgeRDD`s partitioned using the same partitioning strategy.
def innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) => ED3): Ed
```

在大多数的应用中，我们发现，EdgeRDD操作可以通过图操作者(graph operators)或者定义在基本RDD中的操作来完成。

图算法

GraphX包括一组图算法来简化分析任务。这些算法包含在 `org.apache.spark.graphx.lib` 包中，可以被直接访问。

PageRank算法

PageRank度量一个图中每个顶点的重要程度，假定从u到v的一条边代表v的重要性标签。例如，一个Twitter用户被许多其它人粉，该用户排名很高。GraphX带有静态和动态PageRank的实现方法，这些方法在PageRank object中。静态的PageRank运行固定次数 的迭代，而动态的PageRank一直运行，直到收敛。GraphOps允许直接调用这些算法作为图上的方法。

GraphX包含一个我们可以运行PageRank的社交网络数据集的例子。用户集在 `graphx/data/users.txt` 中，用户之间的关系在 `graphx/data/followers.txt` 中。我们通过下面的方法计算 每个用户的PageRank。

```
// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

连通体算法

连通体算法用id标注图中每个连通体，将连通体中序号最小的顶点的id作为连通体的id。例如，在社交网络中，连通体可以近似为集群。GraphX在ConnectedComponents object 中包含了一个算法的实现，我们通过下面的方法计算社交网络数据集中的连通体。

```
/ Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

三角形计数算法

一个顶点有两个相邻的顶点以及相邻顶点之间的边时，这个顶点是一个三角形的一部分。GraphX在TriangleCount object 中实现了一个三角形计数算法，它计算通过每个顶点的三角形的数量。需要注意的是，在计算社交网络数据集的三角形计数时， `TriangleCount` 需要边的方向是规范的方向(srcId < dstId), 并且图通过 `Graph.partitionBy` 分片过。

```
// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt", true).partitionBy(P
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

例子

假定我们想从一些文本文件中构建一个图，限制这个图包含重要的关系和用户，并且在子图上运行page-rank，最后返回与top用户相关的属性。可以通过如下方式实现。

```
// Connect to the Spark cluster
val sc = new SparkContext("spark://master.amplab.org", "research")

// Load my user data and parse into tuples of user id and attribute list
val users = (sc.textFile("graphx/data/users.txt")
  .map(line => line.split(",")).map( parts => (parts.head.toLong, parts.tail) ))

// Parse the edge data which is already in userId -> userId format
val followerGraph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")

// Attach the user attributes
val graph = followerGraph.outerJoinVertices(users) {
  case (uid, deg, Some(attrList)) => attrList
  // Some users may not have attributes so we set them as empty
  case (uid, deg, None) => Array.empty[String]
}

// Restrict the graph to users with usernames and names
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)

// Compute the PageRank
val pagerankGraph = subgraph.pageRank(0.001)

// Get the attributes of the top pagerank users
val userInfoWithPageRank = subgraph.outerJoinVertices(pagerankGraph.vertices) {
  case (uid, attrList, Some(pr)) => (pr, attrList.toList)
  case (uid, attrList, None) => (0.0, attrList.toList)
}

println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkString("\n"))
```

提交应用程序

在Spark bin目录下的 `spark-submit` 可以用来在集群上启动应用程序。它可以通过统一的接口使用Spark支持的所有集群管理器 ，所有你不必为每一个管理器做相应的配置。

用spark-submit启动应用程序

`bin/spark-submit` 脚本负责建立包含Spark以及其依赖的类路径（classpath），它支持不同的集群管理器以及Spark支持的加载模式。

```
./bin/spark-submit \
  --class
  --master  \
  --deploy-mode  \
  --conf = \
  ... # other options
   \
  [application-arguments]
```

一些常用的选项是：

- `--class` ：你的应用程序的入口点(如org.apache.spark.examples.SparkPi)
- `--master` ：集群的master URL(如spark://23.195.26.187:7077)
- `--deploy-mode` ：在worker节点部署你的driver(cluster)或者本地作为外部客户端（client）。默认是client。
- `--conf` ：任意的Spark配置属性，格式是key=value。
- `application-jar` ：包含应用程序以及其依赖的jar包的路径。这个URL必须在集群中全局可见，例如，存在于所有节点的 `hdfs://` 路径或 `file://` 路径
- `application-arguments` ：传递给主类的主方法的参数

一个通用的部署策略是从网关集群提交你的应用程序，这个网关机器和你的worker集群物理上协作。在这种设置下， `client` 模式是适合的。在 `client` 模式下，driver直接在 `spark-submit` 进程 中启动，而这个进程直接作为集群的客户端。应用程序的输入和输出都和控制台相连接。因此，这种模式特别适合涉及REPL的应用程序。

另一种选择，如果你的应用程序从一个和worker机器相距很远的机器上提交，通常情况下用 `cluster` 模式减少drivers和executors的网络迟延。注意， `cluster` 模式目前不支持独立集群、 mesos集群以及python应用程序。

有几个我们使用的集群管理器特有的可用选项。例如，在Spark独立集群的 `cluster` 模式下，你也可以指定 `--supervise` 用来确保driver自动重启（如果它因为非零退出码失败）。 为了列举spark-submit所有的可用选项，用 `--help` 运行它。

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100

# Run on a Spark Standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a Spark Standalone cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster
  --supervise
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000

# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn-cluster \  # can also be `yarn-client` for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000

# Run a Python application on a Spark Standalone cluster
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

Master URLs

传递给Spark的url可以用下面的模式

Master URL	Meaning
local	用一个worker线程本地运行Spark
local[K]	用k个worker线程本地运行Spark(理想情况下，设置这个值为你的机器的核数)
local[*]	用尽可能多的worker线程本地运行Spark
spark://HOST:PORT	连接到给定的Spark独立部署集群master。端口必须是master配置的端口，默认是7077
mesos://HOST:PORT	连接到给定的mesos集群
yarn-client	以 `client` 模式连接到Yarn集群。群集位置将基于通过HADOOP_CONF_DIR变量找到
yarn-cluster	以 `cluster` 模式连接到Yarn集群。群集位置将基于通过HADOOP_CONF_DIR变量找到

Spark独立部署模式

安装Spark独立模式集群

安装Spark独立模式，你只需要将Spark的编译版本简单的放到集群的每个节点。你可以获得每个稳定版本的预编译版本，也可以自己编译。

手动启动集群

你能够通过下面的方式启动独立的master服务器。

```
./sbin/start-master.sh
```

一旦启动，master将会为自己打印出 `spark://HOST:PORT` URL，你能够用它连接到workers或者作为"master"参数传递给 `SparkContext` 。你也可以在master web UI上发现这个URL，master web UI默认的地址是 `http://localhost:8080` 。

相同的，你也可以启动一个或者多个workers或者将它们连接到master。

```
./bin/spark-class org.apache.spark.deploy.worker.Worker spark://IP:PORT
```

一旦你启动了一个worker，查看master web UI。你可以看到新的节点列表以及节点的CPU数以及内存。

下面的配置参数可以传递给master和worker。

Argument	Meaning
-h HOST, --host HOST	监听的主机名
-i HOST, --ip HOST	同上，已经被淘汰
-p PORT, --port PORT	监听的服务的端口（master默认是7077，worker随机）
--webui-port PORT	web UI的端口(master默认是8080，worker默认是8081)
-c CORES, --cores CORES	Spark应用程序可以使用的CPU核数（默认是所有可用）；这个选项仅在worker上可用
-m MEM, --memory MEM	Spark应用程序可以使用的内存数（默认情况是你的机器内存数减去1g）；这个选项仅在worker上可用
-d DIR, --work-dir DIR	用于暂存空间和工作输出日志的目录（默认是SPARK_HOME/work）；这个选项仅在worker上可用
--properties-file FILE	自定义的Spark配置文件的加载目录（默认是conf/spark-defaults.conf）

集群启动脚本

为了用启动脚本启动Spark独立集群，你应该在你的Spark目录下建立一个名为 `conf/slaves` 的文件，这个文件必须包含所有你要启动的Spark worker所在机器的主机名，一行一个。如果 `conf/slaves` 不存在，启动脚本默认为单个机器（localhost），这台机器对于测试是有用的。注意，master机器通过ssh访问所有的worker。在默认情况下，SSH是并行运行，需要设置无密码（采用私有密钥）的访问。 如果你没有设置为无密码访问，你可以设置环境变量 `SPARK_SSH_FOREGROUND` ，为每个worker提供密码。

一旦你设置了这个文件，你就可以通过下面的shell脚本启动或者停止你的集群。

- sbin/start-master.sh：在机器上启动一个master实例
- sbin/start-slaves.sh：在每台机器上启动一个slave实例
- sbin/start-all.sh：同时启动一个master实例和所有slave实例
- sbin/stop-master.sh：停止master实例
- sbin/stop-slaves.sh：停止所有slave实例
- sbin/stop-all.sh：停止master实例和所有slave实例

注意，这些脚本必须在你的Spark master运行的机器上执行，而不是在你的本地机器上面。

你可以在 `conf/spark-env.sh` 中设置环境变量进一步配置集群。利用 `conf/spark-env.sh.template` 创建这个文件，然后将它复制到所有的worker机器上使设置有效。下面的设置可以起作用：

Environment Variable	Meaning
SPARK_MASTER_IP	绑定master到一个指定的ip地址
SPARK_MASTER_PORT	在不同的端口上启动master（默认是7077）
SPARK_MASTER_WEBUI_PORT	master web UI的端口（默认是8080）
SPARK_MASTER_OPTS	应用到master的配置属性，格式是 "-Dx=y"（默认是none），查看下面的表格的选项以组成一个可能的列表
SPARK_LOCAL_DIRS	Spark中暂存空间的目录。包括map的输出文件和存储在磁盘上的RDDs(including map output files and RDDs that get stored on disk)。这必须在一个快速的、你的系统的本地磁盘上。它可以是一个逗号分隔的列表，代表不同磁盘的多个目录
SPARK_WORKER_CORES	Spark应用程序可以用到的核心数（默认是所有可用）
SPARK_WORKER_MEMORY	Spark应用程序用到的内存总数（默认是内存总数减去1G）。注意，每个应用程序个体的内存通过 `spark.executor.memory` 设置
SPARK_WORKER_PORT	在指定的端口上启动Spark worker(默认是随机)
SPARK_WORKER_WEBUI_PORT	worker UI的端口（默认是8081）
SPARK_WORKER_INSTANCES	每台机器运行的worker实例数，默认是1。如果你有一台非常大的机器并且希望运行多个worker，你可以设置这个数大于1。如果你设置了这个环境变量，确保你也设置了 `SPARK_WORKER_CORES` 环境变量用于限制每个worker的核数或者每个worker尝试使用所有的核。
SPARK_WORKER_DIR	Spark worker运行目录，该目录包括日志和暂存空间（默认是SPARK_HOME/work）
SPARK_WORKER_OPTS	应用到worker的配置属性，格式是 "-Dx=y"（默认是none），查看下面表格的选项以组成一个可能的列表
SPARK_DAEMON_MEMORY	分配给Spark master和worker守护进程的内存（默认是512m）
SPARK_DAEMON_JAVA_OPTS	Spark master和worker守护进程的JVM选项，格式是"-Dx=y"（默认为none）
SPARK_PUBLIC_DNS	Spark master和worker公共的DNS名（默认是none）

注意，启动脚本还不支持windows。为了在windows上启动Spark集群，需要手动启动master和workers。

`SPARK_MASTER_OPTS` 支持一下的系统属性：

Property Name	Default	Meaning
spark.deploy.retainedApplications	200	展示完成的应用程序的最大数目。老的应用程序会被删除以满足该限制
spark.deploy.retainedDrivers	200	展示完成的drivers的最大数目。老的应用程序会被删除以满足该限制
spark.deploy.spreadOut	true	这个选项控制独立的集群管理器是应该跨节点传递应用程序还是应努力将程序整合到尽可能少的节点上。在HDFS中，传递程序是数据本地化更好的选择，但是，对于计算密集型的负载，整合会更有效率。
spark.deploy.defaultCores	(infinite)	在Spark独立模式下，给应用程序的默认核数（如果没有设置 `spark.cores.max` ）。如果没有设置，应用程序总数获得所有可用的核，除非设置了 `spark.cores.max` 。在共享集群上设置较低的核数，可用防止用户默认抓住整个集群。
spark.worker.timeout	60	独立部署的master认为worker失败（没有收到心跳信息）的间隔时间。

`SPARK_WORKER_OPTS` 支持的系统属性：

Property Name	Default	Meaning
spark.worker.cleanup.enabled	false	周期性的清空worker/应用程序目录。注意，这仅仅影响独立部署模式。不管应用程序是否还在执行，用于程序目录都会被清空
spark.worker.cleanup.interval	1800 (30分)	在本地机器上，worker清空老的应用程序工作目录的时间间隔
spark.worker.cleanup.appDataTtl	*7 24* 3600 (7天)	每个worker中应用程序工作目录的保留时间。这个时间依赖于你可用磁盘空间的大小。应用程序日志和jar包上传到每个应用程序的工作目录。随着时间的推移，工作目录会很快的填满磁盘空间，特别是如果你运行的作业很频繁。

连接一个应用程序到集群中

为了在Spark集群中运行一个应用程序，简单地传递 `spark://IP:PORT` URL到SparkContext

为了在集群上运行一个交互式的Spark shell，运行一下命令：

```
./bin/spark-shell --master spark://IP:PORT
```

你也可以传递一个选项 `--total-executor-cores <numCores>` 去控制spark-shell的核数。

启动Spark应用程序

spark-submit脚本支持最直接的提交一个Spark应用程序到集群。对于独立部署的集群，Spark目前支持两种部署模式。在 `client` 模式中，driver启动进程与 客户端提交应用程序所在的进程是同一个进程。然而，在 `cluster` 模式中，driver在集群的某个worker进程中启动，只有客户端进程完成了提交任务，它不会等到应用程序完成就会退出。

如果你的应用程序通过Spark submit启动，你的应用程序jar包将会自动分发到所有的worker节点。对于你的应用程序依赖的其它jar包，你应该用 `--jars` 符号指定（如 `--jars jar1,jar2` ）。

另外， `cluster` 模式支持自动的重启你的应用程序（如果程序一非零的退出码退出）。为了用这个特征，当启动应用程序时，你可以传递 `--supervise` 符号到 `spark-submit` 。如果你想杀死反复失败的应用， 你可以通过如下的方式：

```
./bin/spark-class org.apache.spark.deploy.Client kill
```

你可以在独立部署的Master web UI（http://:8080）中找到driver ID。

资源调度

独立部署的集群模式仅仅支持简单的FIFO调度器。然而，为了允许多个并行的用户，你能够控制每个应用程序能用的最大资源数。在默认情况下，它将获得集群的所有核，这只有在某一时刻只 允许一个应用程序才有意义。你可以通过 `spark.cores.max` 在SparkConf中设置核数。

```
val conf = new SparkConf()
             .setMaster(...)
             .setAppName(...)
             .set("spark.cores.max", "10")
val sc = new SparkContext(conf)
```

另外，你可以在集群的master进程中配置 `spark.deploy.defaultCores` 来改变默认的值。在 `conf/spark-env.sh` 添加下面的行：

```
export SPARK_MASTER_OPTS="-Dspark.deploy.defaultCores="
```

这在用户没有配置最大核数的共享集群中是有用的。

高可用

默认情况下，独立的调度集群对worker失败是有弹性的（在Spark本身的范围内是有弹性的，对丢失的工作通过转移它到另外的worker来解决）。然而，调度器通过master去执行调度决定， 这会造成单点故障：如果master死了，新的应用程序就无法创建。为了避免这个，我们有两个高可用的模式。

用ZooKeeper的备用master

利用ZooKeeper去支持领导选举以及一些状态存储，你能够在你的集群中启动多个master，这些master连接到同一个ZooKeeper实例上。一个被选为“领导”，其它的保持备用模式。如果当前 的领导死了，另一个master将会被选中，恢复老master的状态，然后恢复调度。整个的恢复过程大概需要1到2分钟。注意，这个恢复时间仅仅会影响调度新的应用程序-运行在失败master中的 应用程序不受影响。

配置

为了开启这个恢复模式，你可以用下面的属性在`spark-env`中设置`SPARK_DAEMON_JAVA_OPTS`。

System property	Meaning
spark.deploy.recoveryMode	设置ZOOKEEPER去启动备用master模式（默认为none）
spark.deploy.zookeeper.url	zookeeper集群url(如192.168.1.100:2181,192.168.1.101:2181)
spark.deploy.zookeeper.dir	zookeeper保存恢复状态的目录（默认是/spark）

可能的陷阱：如果你在集群中有多个masters，但是没有用zookeeper正确的配置这些masters，这些masters不会发现彼此，会认为它们都是leaders。这将会造成一个不健康的集群状态（因为所有的master都会独立的调度）。

细节

zookeeper集群启动之后，开启高可用是简单的。在相同的zookeeper配置（zookeeper URL和目录）下，在不同的节点上简单地启动多个master进程。master可以随时添加和删除。

为了调度新的应用程序或者添加worker到集群，它需要知道当前leader的IP地址。这可以通过简单的传递一个master列表来完成。例如，你可能启动你的SparkContext指向 `spark://host1:port1,host2:port2` 。这将造成你的SparkContext同时注册这两个master-如果 `host1` 死了，这个配置文件将一直是正确的，因为我们将找到新的leader- `host2` 。

"registering with a Master"和正常操作之间有重要的区别。当启动时，一个应用程序或者worker需要能够发现和注册当前的leader master。一旦它成功注册，它就在系统中了。如果错误发生，新的leader将会接触所有之前注册的应用程序和worker，通知他们领导关系的变化，所以它们甚至不需要事先知道新启动的leader的存在。

由于这个属性的存在，新的master可以在任何时候创建。你唯一需要担心的问题是新的应用程序和workers能够发现它并将它注册进来以防它成为leader master。

用本地文件系统做单节点恢复

zookeeper是生产环境下最好的选择，但是如果你想在master死掉后重启它， `FILESYSTEM` 模式可以解决。当应用程序和worker注册，它们拥有足够的状态写入提供的目录，以至于在重启master 进程时它们能够恢复。

配置

为了开启这个恢复模式，你可以用下面的属性在 `spark-env` 中设置 `SPARK_DAEMON_JAVA_OPTS` 。

System property	Meaning
spark.deploy.recoveryMode	设置为FILESYSTEM开启单节点恢复模式（默认为none）
spark.deploy.recoveryDirectory	用来恢复状态的目录

细节

- 这个解决方案可以和监控器/管理器（如monit）相配合，或者仅仅通过重启开启手动恢复。
- 虽然文件系统的恢复似乎比没有做任何恢复要好，但对于特定的开发或实验目的，这种模式可能是次优的。特别是，通过 `stop-master.sh` 杀掉master不会清除它的恢复状态，所以，不管你何时启动一个新的master，它都将进入恢复模式。这可能使启动时间增加到1分钟。
- 虽然它不是官方支持的方式，你也可以创建一个NFS目录作为恢复目录。如果原始的master节点完全死掉，你可以在不同的节点启动master，它可以正确的恢复之前注册的所有应用程序和workers。未来的应用程序会发现这个新的master。

在YARN上运行Spark

配置

大部分为 `Spark on YARN` 模式提供的配置与其它部署模式提供的配置相同。下面这些是为 `Spark on YARN` 模式提供的配置。

Spark属性

Property Name	Default	
spark.yarn.applicationMaster.waitTries	10	ApplicationMaster等待S 数
spark.yarn.submit.file.replication	HDFS默认的复制次数（3）	上传到HDFS的文件的H 任何分布式缓存文件/档
spark.yarn.preserve.staging.files	false	设置为true，则在作业结 布式缓存文件）而不是册
spark.yarn.scheduler.heartbeat.interval-ms	5000	Spark application maste （ms）
spark.yarn.max.executor.failures	numExecutors * 2,最小为3	失败应用程序之前最大的
spark.yarn.historyServer.address	(none)	Spark历史服务器（如ho 式（http://）。默认情况 Spark应用程序完成从R 这个地址从YARN Reso
spark.yarn.dist.archives	(none)	提取逗号分隔的档案列表
spark.yarn.dist.files	(none)	放置逗号分隔的文件列表
spark.yarn.executor.memoryOverhead	executorMemory * 0.07,最小384	分配给每个执行器的堆内 符串或者其它本地开销 况下是6%-10%）
spark.yarn.driver.memoryOverhead	driverMemory * 0.07,最小384	分配给每个driver的堆内 符串或者其它本地开销 况下是6%-10%）
spark.yarn.queue	default	应用程序被提交到的YA
spark.yarn.jar	(none)	Spark jar文件的位置，覆 到本地安装的Spark jar。 许YARN缓存它到节点上 HDFS中的jar包，可以这
spark.yarn.access.namenodes	(none)	你的Spark应用程序访问 如， `spark.yarn.access.` Spark应用程序必须访问 们。Spark获得namenod 程的HDFS集群。
spark.yarn.containerLauncherMaxThreads	25	为了启动执行者容器，应
spark.yarn.appMasterEnv.[EnvironmentVariableName]	(none)	添加通过 `EnvironmentVa` YARN上的启动。用户可 cluster模式下，这控制S 执行器启动者的环境。

在YARN上启动Spark

确保 `HADOOP_CONF_DIR` 或 `YARN_CONF_DIR` 指向的目录包含Hadoop集群的（客户端）配置文件。这些配置用于写数据到dfs和连接到YARN ResourceManager。

有两种部署模式可以用来在YARN上启动Spark应用程序。在yarn-cluster模式下，Spark driver运行在application master进程中，这个进程被集群中的YARN所管理，客户端会在初始化应用程序 之后关闭。在yarn-client模式下，driver运行在客户端进程中，application master仅仅用来向YARN请求资源。

和Spark单独模式以及Mesos模式不同，在这些模式中，master的地址由"master"参数指定，而在YARN模式下，ResourceManager的地址从Hadoop配置得到。因此master参数是简单的 `yarn-client` 和 `yarn-cluster` 。

在yarn-cluster模式下启动Spark应用程序。

```
./bin/spark-submit --class path.to.your.Class --master yarn-cluster [options]  [app optio
```

例子：

```
$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master yarn-cluster \
    --num-executors 3 \
    --driver-memory 4g \
    --executor-memory 2g \
    --executor-cores 1 \
    --queue thequeue \
    lib/spark-examples*.jar \
    10
```

以上启动了一个YARN客户端程序用来启动默认的 Application Master，然后SparkPi会作为Application Master的子线程运行。客户端会定期的轮询Application Master用于状态更新并将更新显示在控制台上。一旦你的应用程序运行完毕，客户端就会退出。

在yarn-client模式下启动Spark应用程序，运行下面的shell脚本

```
$ ./bin/spark-shell --master yarn-client
```

添加其它的jar

在yarn-cluster模式下，driver运行在不同的机器上，所以离开了保存在本地客户端的文件， `SparkContext.addJar` 将不会工作。为了使 `SparkContext.addJar` 用到保存在客户端的文件， 在启动命令中加上 `--jars` 选项。

```
$ ./bin/spark-submit --class my.main.Class \
    --master yarn-cluster \
    --jars my-other-jar.jar,my-other-other-jar.jar
    my-main-jar.jar
    app_arg1 app_arg2
```

注意事项

- 在Hadoop 2.2之前，YARN不支持容器核的资源请求。因此，当运行早期的版本时，通过命令行参数指定的核的数量无法传递给YARN。在调度决策中，核请求是否兑现取决于用哪个调度器以及 如何配置调度器。
- Spark executors使用的本地目录将会是YARN配置（yarn.nodemanager.local-dirs）的本地目录。如果用户指定了 `spark.local.dir` ，它将被忽略。
- `--files` 和 `--archives` 选项支持指定带 # 号文件名。例如，你能够指定 `--files localtest.txt#appSees.txt` ，它上传你在本地命名为 `localtest.txt` 的文件到HDFS，但是将会链接为名称 `appSees.txt` 。当你的应用程序运行在YARN上时，你应该使用 `appSees.txt` 去引用该文件。
- 如果你在yarn-cluster模式下运行 `SparkContext.addJar` ，并且用到了本地文件， `--jars` 选项允许 `SparkContext.addJar` 函数能够工作。如果你正在使用 HDFS, HTTP, HTTPS或FTP，你不需要用到该选项

Spark配置

Spark提供三个位置用来配置系统：

- Spark properties控制大部分的应用程序参数，可以用SparkConf对象或者java系统属性设置
- Environment variables可以通过每个节点的 `conf/spark-env.sh` 脚本设置每台机器的设置。例如IP地址
- Logging可以通过log4j.properties配置

Spark属性

Spark属性控制大部分的应用程序设置，并且为每个应用程序分别配置它。这些属性可以直接在SparkConf上配置，然后传递给 `SparkContext` 。 `SparkConf` 允许你配置一些通用的属性（如master URL、应用程序明）以及通过 `set()` 方法设置的任意键值对。例如，我们可以用如下方式创建一个拥有两个线程的应用程序。注意，我们用 `local[2]` 运行，这意味着两个线程-表示最小的 并行度，它可以帮助我们检测当在分布式环境下运行的时才出现的错误。

```
val conf = new SparkConf()
             .setMaster("local[2]")
             .setAppName("CountingSheep")
             .set("spark.executor.memory", "1g")
val sc = new SparkContext(conf)
```

注意，我们在本地模式中拥有超过1个线程。和Spark Streaming的情况一样，我们可能需要一个线程防止任何形式的饥饿问题。

动态加载Spark属性

在一些情况下，你可能想在 `SparkConf` 中避免硬编码确定的配置。例如，你想用不同的master或者不同的内存数运行相同的应用程序。Spark允许你简单地创建一个空conf。

```
val sc = new SparkContext(new SparkConf())
```

然后你在运行时提供值。

```
./bin/spark-submit --name "My app" --master local[4] --conf spark.shuffle.spill=false
  --conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myA
```

Spark shell和 `spark-submit` 工具支持两种方式动态加载配置。第一种方式是命令行选项，例如 `--master` ，如上面shell显示的那样。 `spark-submit` 可以接受任何Spark属性，用 `--conf` 标记表示。但是那些参与Spark应用程序启动的属性要用特定的标记表示。运行 `./bin/spark-submit --help` 将会显示选项的整个列表。

`bin/spark-submit` 也会从 `conf/spark-defaults.conf` 中读取配置选项，这个配置文件中，每一行都包含一对以空格分开的键和值。例如：

```
spark.master            spark://5.6.7.8:7077
spark.executor.memory   512m
spark.eventLog.enabled  true
spark.serializer        org.apache.spark.serializer.KryoSerializer
```

任何标签（flags）指定的值或者在配置文件中的值将会传递给应用程序，并且通过 `SparkConf` 合并这些值。在 `SparkConf` 上设置的属性具有最高的优先级，其次是传递给 `spark-submit` 或者 `spark-shell` 的属性值，最后是 `spark-defaults.conf` 文件中的属性值。

查看Spark属性

在 `http://<driver>:4040` 上的应用程序web UI在“Environment”标签中列出了所有的Spark属性。这对你确保设置的属性的正确性是很有用的。注意，只有通过spark-defaults.conf, SparkConf以及 命令行直接指定的值才会显示。对于其它的配置属性，你可以认为程序用到了默认的值。

可用的属性

控制内部设置的大部分属性都有合理的默认值，一些最通用的选项设置如下：

应用程序属性

Property Name	Default	
spark.app.name	(none)	你的应用程序 中出现
spark.master	(none)	集群管理器
spark.executor.memory	512m	每个executo 存串拥有相
spark.driver.memory	512m	driver进程使
spark.driver.maxResultSize	1g	每个Spark a 化结果的总 1m，0代表 限制，工作 driver出现内 spark.driver 耗）。设置 溢出错误。
spark.serializer	org.apache.spark.serializer.JavaSerializer	序列化对象 可以序列化 很慢。所有 用org.apach
spark.kryo.classesToRegister	(none)	如果你用Kr 自定义类名
spark.kryo.registrator	(none)	如果你用Kr 的自定义类 册你的类， 则 `spark.kr` 它应该设置
spark.local.dir	/tmp	Spark中暂存 以及更高的 SPARK_LO 和LOCAL_D
spark.logConf	false	当SparkCon SparkConf

运行环境

Property Name	Default	Me
spark.executor.extraJavaOptions	(none)	传递给executors的JV 置或者其它日志设置。 Spark属性或者堆大小 要用SparkConf对象或 的 `spark-defaults.conf` 通过 `spark.executor.me`
		附加到executors的clas

spark.executor.extraClassPath	(none)	实体。这个设置存在的 本的向后兼容问题。用
spark.executor.extraLibraryPath	(none)	指定启动executor的JV
spark.executor.logs.rolling.strategy	(none)	设置executor日志的滚 下没有开启。可以配置 动）和 `size` (基于大小 用 `spark.executor.logs` 置滚动间隔；对于 `size` 用 `spark.executor.logs` 置最大的滚动大小
spark.executor.logs.rolling.time.interval	daily	executor日志滚动的时 开启。合法的值是 `dail` 及任意的秒。
spark.executor.logs.rolling.size.maxBytes	(none)	executor日志的最大滚 开启。值设置为字节
spark.executor.logs.rolling.maxRetainedFiles	(none)	设置被系统保留的最近 老的日志文件将被删除
spark.files.userClassPathFirst	false	(实验性)当在Executors 加的jar比Spark自己的j 以降低Spark依赖和用 是一个实验性的特征。
spark.python.worker.memory	512m	在聚合期间，每个pyth 数。在聚合期间，如果 将会将数据塞进磁盘中
spark.python.profile	false	在Python worker中开启 过 `sc.show_profiles()` driver退出前展示分析 过 `sc.dump_profiles(pa` 中。如果一些分析结果 driver退出前，它们再
spark.python.profile.dump	(none)	driver退出前保存分析 每个RDD都会分别dum 过 `ptats.Stats()` 加载 个属性，分析结果不会
spark.python.worker.reuse	true	是否重用python worke 数量的Python workers fork()一个Python进程。 播，这个设置将非常有 每个任务从JVM到Pyth
spark.executorEnv.[EnvironmentVariableName]	(none)	通过 `EnvironmentVariab` 量到executor进程。用 个 `EnvironmentVariable`
spark.mesos.executor.home	driver side	设置安装在Mesos的ex 录。默认情况下，exec Spark本地（home）目

		见。注意，如果没有通 定Spark的二进制包，这
spark.mesos.executor.memoryOverhead	executor memory * 0.07, 最小384m	这个值是 `spark.executo` 计算mesos任务的总内 硬编码设置。最后的值 择 `spark.mesos.executo` 者 `spark.executor.memo`

Shuffle行为(Behavior)

Property Name	Default	Meaning
spark.shuffle.consolidateFiles	false	如果设置为"true"，在shuffle期间，合并的中间文件将会被创建。创建更少的文件可以提供文件系统的shuffle的效率。这些shuffle都伴随着大量递归任务。当用ext4和dfs文件系统时，推荐设置为"true"。在ext3中，因为文件系统的限制，这个选项可能机器（大于8核）降低效率
spark.shuffle.spill	true	如果设置为"true"，通过将多出的数据写入磁盘来限制内存数。通过 `spark.shuffle.memoryFraction` 来指定spilling的阈值
spark.shuffle.spill.compress	true	在shuffle时，是否将spilling的数据压缩。压缩算法通过 `spark.io.compression.codec` 指定。
spark.shuffle.memoryFraction	0.2	如果 `spark.shuffle.spill` 为“true”，shuffle中聚合和合并组操作使用的java堆内存占总内存的比重。在任何时候，shuffles使用的所有内存内maps的集合大小都受这个限制的约束。超过这个限制，spilling数据将会保存到磁盘上。如果spilling太过频繁，考虑增大这个值
spark.shuffle.compress	true	是否压缩map操作的输出文件。一般情况下，这是一个好的选择。
spark.shuffle.file.buffer.kb	32	每个shuffle文件输出流内存内缓存的大小，单位是kb。这个缓存减少了创建只中间shuffle文件中磁盘搜索和系统访问的数量
spark.reducer.maxMbInFlight	48	从递归任务中同时获取的map输出数据的最大大小（mb）。因为每一个输出都需要我们创建一个缓存用

spark.reducer.maxMbInFlight	48	个输出都需要我们创建一个缓存用来接收，这个设置代表每个任务固定的内存上限，所以除非你有更大的内存，将其设置小一点
spark.shuffle.manager	sort	它的实现用于shuffle数据。有两种可用的实现： sort 和 hash 。基于sort的shuffle有更高的内存使用率
spark.shuffle.sort.bypassMergeThreshold	200	(Advanced) In the sort-based shuffle manager, avoid merge-sorting data if there is no map-side aggregation and there are at most this many reduce partitions
spark.shuffle.blockTransferService	netty	实现用来在executor直接传递shuffle和缓存块。有两种可用的实现： netty 和 nio 。基于netty的块传递在具有相同的效率情况下更简单

Spark UI

Property Name	Default	Meaning
spark.ui.port	4040	你的应用程序dashboard的端口。显示内存和工作量数据
spark.ui.retainedStages	1000	在垃圾回收之前，Spark UI和状态API记住的stage数
spark.ui.retainedJobs	1000	在垃圾回收之前，Spark UI和状态API记住的job数
spark.ui.killEnabled	true	运行在web UI中杀死stage和相应的job
spark.eventLog.enabled	false	是否记录Spark的事件日志。这在应用程序完成后，重新构造web UI是有用的
spark.eventLog.compress	false	是否压缩事件日志。需要 spark.eventLog.enabled 为true
spark.eventLog.dir	file:///tmp/spark-events	Spark事件日志记录的基本目录。在这个基本目录下，Spark为每个应用程序创建一个子目录。各个应用程序记录日志到直到的目录。用户可能想设置这为统一的地点，像HDFS一样，所以历史文件可以通过历史服务器读取

压缩和序列化

Property Name	Default	
spark.broadcast.compress	true	在
spark.rdd.compress	true	是
spark.io.compression.codec	snappy	压 用 定
spark.io.compression.snappy.block.size	32768	S
spark.io.compression.lz4.block.size	32768	L
spark.closure.serializer	org.apache.spark.serializer.JavaSerializer	闭
spark.serializer.objectStreamReset	100	当 必 下
spark.kryo.referenceTracking	true	当 率
spark.kryo.registrationRequired	false	是 个
spark.kryoserializer.buffer.mb	0.064	K 么
spark.kryoserializer.buffer.max.mb	64	K

Networking

Property Name	Default	Meaning
spark.driver.host	(local hostname)	driver监听的主机名或者IP地址。这用于和executors以及独立的master通信
spark.driver.port	(random)	driver监听的接口。这用于和executors以及独立的master通信
spark.fileserver.port	(random)	driver的文件服务器监听的端口
spark.broadcast.port	(random)	driver的HTTP广播服务器监听的端口
spark.replClassServer.port	(random)	driver的HTTP类服务器监听的端口
spark.blockManager.port	(random)	块管理器监听的端口。这些同时存在于driver和executors
spark.executor.port	(random)	executor监听的端口。用于与driver通信
spark.port.maxRetries	16	当绑定到一个端口，在放弃前重试的最大次数
spark.akka.frameSize	10	在"control plane"通信中允许的最大消息大小。如果你的任务需要发送大的结果到

		driver中，调大这个值
spark.akka.threads	4	通信的actor线程数。当driver有很多CPU核时，调大它是有用的
spark.akka.timeout	100	Spark节点之间的通信超时。单位是s
spark.akka.heartbeat.pauses	6000	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). Acceptable heart beat pause in seconds for akka. This can be used to control sensitivity to gc pauses. Tune this in combination of `spark.akka.heartbeat.interval` and `spark.akka.failure-detector.threshold` if you need to.
spark.akka.failure-detector.threshold	300.0	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). This maps to akka's `akka.remote.transport-failure-detector.threshold`. Tune this in combination of `spark.akka.heartbeat.pauses` and `spark.akka.heartbeat.interval` if you need to.
spark.akka.heartbeat.interval	1000	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). A larger interval value in seconds reduces network overhead and a smaller value (~ 1 s) might be more informative for akka's failure detector. Tune this in combination of `spark.akka.heartbeat.pauses` and `spark.akka.failure-detector.threshold` if you need to. Only positive use case for using failure detector can be, a sensistive failure detector can help evict rogue executors really quick. However this is usually not the case as gc pauses and network lags are expected in a real Spark cluster. Apart from that enabling this leads to a lot of exchanges of heart beats between nodes leading to flooding the network with those.

Security

Property Name	Default	Meaning
spark.authenticate	false	是否Spark验证其内部连接。如果不 看 `spark.authenticate.secret`
spark.authenticate.secret	None	设置Spark两个组件之间的密匙验证 上，但是需要验证，这个选项必须设
spark.core.connection.auth.wait.timeout	30	连接时等待验证的实际。单位为秒
spark.core.connection.ack.wait.timeout	60	连接等待回答的时间。单位为秒。为 你可以设置更大的值

spark.ui.filters	None	应用到Spark web UI的用于过滤类名
		滤器必须是标准的javax servlet Filte
		性也可以指定每个过滤器的参数。 `s`
		`filter>.params='param1=value1,param`
		`Dspark.ui.filters=com.test.filter1`
		`Dspark.com.test.filter1.params='par`

spark.acls.enable	false	是否开启Spark acls。如果开启了， 去查看或修改job。 Note this require so if the user comes across as null 利用使用过滤器验证和设置用户
spark.ui.view.acls	empty	逗号分隔的用户列表，列表中的用户 UI的权限。默认情况下，只有启动S 限
spark.modify.acls	empty	逗号分隔的用户列表，列表中的用户 限。默认情况下，只有启动Spark jo
spark.admin.acls	empty	逗号分隔的用户或者管理员列表，列 查看和修改所有Spark job的权限。如 群，有一组管理员或开发者帮助deb

Spark Streaming

Property Name	Default	Meaning
spark.streaming.blockInterval	200	在这个时间间隔（ms）内，通过Spark Streaming receivers接收的数据在保存到Spark之前，chunk为数据块。推荐的最小值为50ms
spark.streaming.receiver.maxRate	infinite	每秒钟每个receiver将接收的数据的最大记录数。有效的情况下，每个流将消耗至少这个数目的记录。设置这个配置为0或者-1将会不作限制
spark.streaming.receiver.writeAheadLogs.enable	false	Enable write ahead logs for receivers. All the input data received through receivers will be saved to write ahead logs that will allow it to be recovered after driver failures
spark.streaming.unpersist	true	强制通过Spark Streaming生成并持久化的RDD自动从Spark内存中非持久化。通过Spark Streaming接收的原始输入数据也将清除。设置这个属性为false允许流应用程序访问原始数据和持久化RDD，因为它们没有被自动清除。但是它会造成更高的内存花费

环境变量

通过环境变量配置确定的Spark设置。环境变量从Spark安装目录下的 `conf/spark-env.sh` 脚本读取（或者windows的 `conf/spark-env.cmd` ）。在独立的或者Mesos模式下，这个文件可以给机器 确定的信息，如主机名。当运行本地应用程序或者提交脚本时，它也起作用。

注意，当Spark安装时， `conf/spark-env.sh` 默认是不存在的。你可以复制 `conf/spark-env.sh.template` 创建它。

可以在 `spark-env.sh` 中设置如下变量：

Environment Variable	Meaning
JAVA_HOME	java安装的路径
PYSPARK_PYTHON	PySpark用到的Python二进制执行文件路径
SPARK_LOCAL_IP	机器绑定的IP地址
SPARK_PUBLIC_DNS	你Spark应用程序通知给其他机器的主机名

除了以上这些，Spark standalone cluster scripts也可以设置一些选项。例如 每台机器使用的核数以及最大内存。

因为 `spark-env.sh` 是shell脚本，其中的一些可以以编程方式设置。例如，你可以通过特定的网络接口计算 `SPARK_LOCAL_IP` 。

配置Logging

Spark用log4j logging。你可以通过在conf目录下添加 `log4j.properties` 文件来配置。一种方法是复制 `log4j.properties.template` 文件。